

FILE COPY

1

AD-A213 499

A SIMULATION MODEL FOR DYNAMIC SYSTEM
AVAILABILITY ANALYSIS

by

DISTER LEROY DEOSS, JR.

B.S., Electrical Engineering
University of Texas, Austin
(1984)

DTIC
ELECTED
OCT 12 1989

S D C

SUBMITTED TO THE DEPARTMENT OF
NUCLEAR ENGINEERING
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF SCIENCE IN NUCLEAR ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1989

c Dister LeRoy Deoss, Jr., 1989

The author hereby grants to M.I.T. and to the U. S. Government
permission to reproduce and to distribute copies of this thesis
document in whole or in part.

Signature of Author.....*Dister L. Deoss*.....

Department of Nuclear Engineering
May 12, 1989

Certified by.....*Nathan O. Siu*.....

Nathan O. Siu
Professor of Nuclear Engineering
Thesis Supervisor

Accepted by.....*Allan F. Henry*.....

Allan F. Henry
Chairman, Department Committee on Graduate Students

| |
|-----------------------------|
| DISTRIBUTION STATEMENT A |
| Approved for public release |
| Distribution Unlimited |

89 10 10129

A SIMULATION MODEL FOR DYNAMIC SYSTEM
AVAILABILITY ANALYSIS

by

DISTER LEROY DEOSS, JR.

Submitted to the Department of Nuclear Engineering
on May 12, 1989 in partial fulfillment of the
requirements for the degree of Master of Science in
Nuclear Engineering

ABSTRACT

Current methods of system reliability analysis cannot easily evaluate the time dependent availability of complex dynamic systems. Improved methods are needed to treat such issues as process variables, feedback, and rule based interactions between components.

A dynamic Monte Carlo system availability simulation model is developed. The basic model, called DYMCA, is based on three fundamental modeling objectives. First, to provide the ability to analyze time-dependent availability of dynamic systems. Second, to provide a model which is easy to apply and interpret. And third, to create a model which can easily be modified to incorporate additional features as needed. The output generated by the program includes time-dependent system unavailability information and average system unavailability over the duration of the simulated time period.

The DYMCA model is tested on several basic availability analysis problems to demonstrate program capabilities. These tests include a single component with exponential failure and repair times, a single component with two repair states, a two-out-of-three pump failure system, and a phased mission problem requiring the forced change of a system component state after the start of the analysis. A modification of the DYMCA program was also developed to demonstrate the capability of treating continuous process variables in a dynamic simulation model.

Results of all test were compared with analytical results where possible, and with Markovian analysis techniques in other cases. The simulation model provided accurate unavailability results on all example problems tested. Further work needs to be done to expand the capabilities of the basic DYMCA model and to continue program testing on more complex problems.

Thesis Supervisor: Nathan O. Siu

Title: Assistant Professor of Nuclear Engineering

ACKNOWLEDGEMENTS

My deepest appreciation is given to my thesis supervisor, Professor Nathan Siu. Without his instruction and advice this work would not have been possible. He provided the basic groundwork for the simulation model developed in this thesis, and was always available to provide guidance when answers to problems seemed difficult to find. I also wish to acknowledge Professor Tunc Aldemir of Ohio State University for providing data necessary for one of the example problems treated.

The U.S. Navy has provided the funding for my education, and I sincerely wish to thank the Navy for giving me this opportunity. Four more years of service is indeed a small price to pay for the opportunity to attend such a prestigious university. I only hope that this work may in some way prove beneficial to the Navy.

Lastly, I wish to thank my family and friends who provided encouragement and support when I needed it most. And a special thanks is given to my fiancee, Laurie, who helped when should could and patiently endured through countless hours of watching me type on the computer without once uttering a word of complaint.

| | |
|--|----------------|
| Accesion For | |
| NTIS CRAGI <input checked="" type="checkbox"/> | |
| DTIC TAB <input type="checkbox"/> | |
| Unannounced <input type="checkbox"/> | |
| Justification | |
| By <i>Per form 50</i> | |
| Distribution | |
| Acquisition Source | |
| Dist | Acquisition Of |
| A-1 | |

Table of Contents

| | Page |
|--|------|
| Abstract | 2 |
| Acknowledgements | 3 |
| Table of Contents | 4 |
| List of Figures | 6 |
| List of Tables | 8 |
| Chapter 1. Introduction | |
| 1.1 Foreword | 9 |
| 1.2 Background | 10 |
| 1.3 Organization of this Work | 11 |
| Chapter 2. A Survey of Current System Reliability Analysis Techniques | |
| 2.1 Introduction | 13 |
| 2.2 Static Methods | |
| 2.2.1 Fault Trees | 13 |
| 2.2.2 GO Methodology..... | 19 |
| 2.3 Dynamic Methods | |
| 2.3.1 Event Trees | 25 |
| 2.3.2 Digraphs | 27 |
| 2.3.3 GO-FLOW Methodology | 29 |
| 2.3.4 Markovian Analysis | 37 |
| 2.3.5 Monte Carlo Simulation | 42 |
| 2.4 Chapter Summary | 46 |
| Chapter 3. DYMCA M Dynamic Simulation Model | |
| 3.1 Introduction | 48 |
| 3.2 Simulation Language | |
| 3.2.1 An Overview | 49 |
| 3.2.2 SIMSCRIPT II.5 | 59 |
| 3.3 Program Objectives | 65 |
| 3.4 Model Assumptions | 68 |
| 3.5 Program Description | 77 |
| 3.6 Chapter Summary | 87 |
| Chapter 4. Test Runs and Results | |
| 4.1 Introduction | 89 |
| 4.2 Single Component, Single Repair State | 90 |
| 4.3 Single Component, Dual Repair State | 98 |
| 4.4 Two out of Three Pumps | 103 |
| 4.5 GO-FLOW Example Problem | 112 |
| 4.6 Chapter Summary | 118 |

Table of Contents (continued)**Chapter 5. Continuous Simulation TANK Program**

| | |
|--|-----|
| 5.1 Introduction | 121 |
| 5.2 Problem Description | 122 |
| 5.3 The TANK Program Modifications to DYMCA..... | 128 |
| 5.4 TANK Results | |
| 5.4.1 Analysis of Case A | 138 |
| 5.4.2 Analysis of Case F | 144 |
| 5.4.3 Simulation Analysis | 149 |
| 5.5 Chapter Summary | 163 |

Chapter 6. Summary and Conclusions

| | |
|---|-----|
| 6.1 Discussion of Methods | 166 |
| 6.2 Discussion of Results | 169 |
| 6.3 Strengths and Weaknesses | 171 |
| 6.4 Conclusions and Recommendations | 173 |

| | |
|------------------|-----|
| References | 175 |
|------------------|-----|

| | |
|--|-----|
| Appendix A. DYMCA Input File Description | 178 |
|--|-----|

| | |
|---|-----|
| Appendix B. DYMCA Program Listing | 190 |
|---|-----|

| | |
|--|-----|
| Appendix C. TANK Program Listing | 243 |
|--|-----|

| | |
|--------------------------------------|-----|
| Appendix D. Sample Input Files | 256 |
|--------------------------------------|-----|

| | |
|---------------------------------------|-----|
| Appendix E. Sample Output Files | 261 |
|---------------------------------------|-----|

List of Figures

| | Page |
|--|------|
| 2.1 System Reliability Analysis Methods | 14 |
| 2.2 Fault Tree Example Problem | 16 |
| 2.3 Fault Tree Limitation Example | 18 |
| 2.4 GO Operators | 21 |
| 2.5 GO Example Problem | 24 |
| 2.6 Event Tree Example | 26 |
| 2.7 GO-FLOW Operators | 31 |
| 2.8 GO-FLOW Chart Example | 34 |
| 3.1 General Component Model | 52 |
| 3.2 Component History | 53 |
| 3.3 Limited Repair Resource Modeling | 64 |
| 3.4 Event Scheduling Approaches | 66 |
| 3.5 Active Component | 71 |
| 3.6 Passive Component | 72 |
| 3.7 Valve | 73 |
| 3.8 Check Valve | 74 |
| 3.9 Switch | 75 |
| 3.10 DYMCA Program Flow Chart | 80 |
| 4.1 Simulation Unavailability Time Line | 92 |
| 4.2 Single Component, Single Repair State Average Unavailability | 95 |
| 4.3 Single Component, Single Repair State Time Dependent Unavailability | 97 |
| 4.4 Single Component, Dual Repair State Average Unavailability | 101 |

List of Figures (continued)

| | |
|--|-----|
| 4.5 Single Component, Dual Repair State | |
| Time Dependent Unavailability | 102 |
| 4.6 Two Out of Three Pumps System Diagram | 105 |
| 4.7 Two Out Of Three Component Average Unavailability | 107 |
| 4.8 Markov State Transition Diagram for | |
| Two Out of Three System | 109 |
| 4.9 Two Out Of Three Component Time Dependent Unavailability . | 110 |
| 4.10 Light Bulb Problem Diagram | 113 |
| 5.1 Tank Problem Diagram | 123 |
| 5.2 Flow Chart for TANK Program | 131 |
| 5.3 TANK Program Signals | 135 |
| 5.4 Tank Case A State Transition Diagram | 139 |
| 5.5 Tank Case F State Transition Diagram | 151 |
| 5.6 Case A - Cumulative Dryout Probability | 153 |
| 5.7 Case A - Cumulative Overflow Probability | 154 |
| 5.8 Case F - Cumulative Dryout Probability | 159 |
| 5.9 Case F - Cumulative Overflow Probability | 160 |
| 5.10 Comparison with Aldemir's Results for Case A | 161 |
| 5.11 Comparison with Aldemir's Results for Case F | 162 |
| A.1 Example DYMCA Input File | 179 |

List of Tables

| | Page |
|---|------|
| 2.1 Summary of the Function of GO-FLOW operators | 32 |
| 2.2 Operators Used in GO-FLOW Sample Problem | 36 |
| 2.3 Signals Defined in Sample GO-FLOW Problem | 36 |
| 2.4 Calculation Steps for Sample Problem | 37 |
| 3.1 Advantages and Disadvantages of Simulation | 50 |
| 3.2 Comparison of Languages for Discrete-Event Simulation | 58 |
| 3.3 DYMCA Subroutines | 79 |
| 4.1 Single Component, Single Repair State Instantaneous Unavailability | 94 |
| 4.2 Single Component, Dual Repair State Instantaneous Unavailability | 100 |
| 4.3 Two Out Of Three Component Instantaneous Unavailability .. | 106 |
| 4.4 Light Bulb Problem Results (1,000 to 5,000 trials) | 115 |
| 4.5 Light Bulb Problem Results (6,000 to 10,000 trials) | 116 |
| 5.1 Flow Control Unit States as a Function of Fluid Level | 124 |
| 5.2 TANK Subroutines | 129 |
| 5.3 Case A Failure Sequence Summary | 148 |
| 5.4 Case F Failure Sequence Summary | 149 |
| 5.5 Markov States for Tank Case A | 150 |
| 5.6 Markov Equations for Tank Case A | 152 |
| 5.7 Markov States for Tank Case F | 156 |
| 5.8 Markov Equations for Tank Case F | 157 |

Chapter 1

Introduction

1.1 Foreword

The Engineering community has always depended upon the methods of system reliability evaluation and prediction to solve practical engineering problems. The use of all engineered structures and inventions is dependent upon their ability to perform to some predetermined specifications. Thus as technology advances and systems become more complex it becomes necessary to derive new and better ways to ensure the reliability of such systems.

Recent examples provide abundant evidence of the need for proper attention to reliability analysis in engineering design. One prominent case is the failure of a seal on the booster rocket for the space shuttle Challenger in January of 1986, which lead to the deaths of five astronauts and a three year delay in the NASA space shuttle program. In the nuclear industry, the failure of a pressure operated relief valve to reseat can be argued to be at least partially the cause for the melt down of the core of the unit 2 reactor at Three Mile Island in March 1979. And there are many other such events in all engineering disciplines which indicate dramatically the results of engineering systems which have not performed adequately.

The field of reliability analysis has continued to meet the challenge in the increasingly complex technology of today's society. Over the past two decades many advances have been made in all areas of system analysis and progress continues to be made. To meet the needs of future technological advances and to provide for the highest possible levels of safety and reliability in all aspects of engineering it is necessary for systems reliability analysts to continue to improve the state of the art by

identifying and developing new analysis and prediction techniques.

1.2 Background

Today many approaches and methods are employed in the process of system reliability analysis. For single components which are mass produced and essentially identical, fundamental probability laws are applied to estimate the probability of any given component functioning correctly for a specified number of hours. This estimate is made based on historical performance of identical component and engineering judgement about any improvements which may have been made.

For systems made of many components, fault trees or event trees may be used to calculate static system reliability characteristics as described in references D-1, M-1, and P-1. From these trees, minimal cut sets are identified which can be evaluated numerically to provide failure rate information for the system. For complex systems where it is necessary to determine all combinations of conditions which may lead to a specified deviation of a system parameter, digraph methods may be used as discussed in reference K-1. Then fault tree synthesis methods may be used to construct fault trees from the digraph.

For repairable systems with exponential repair and failure rates, Markovian analysis may be used to compute time dependent system availability and unavailability as delineated in references M-1, P-1, and G-2. Markov systems may be solved explicitly using Laplace transforms, they may be solved by computing eigenvalues and eigenvectors, or they may be solved by computer numerical integration techniques. Through use of Chapman-Kolmogorov equations it is possible to determine the probability of transition between any two system states given that the probabilities of all intermediate transitions are known.

The current methods available can be characterized as belonging to one of two major categories. These are static reliability analysis methods, and dynamic methods. The former are useful in determining system reliability at a specified instant of time. The latter give time dependent system information in either discrete or continuous form. Static methods can give detailed information about a system at a specific time point, but are often not useful in evaluating dynamic systems. Dynamic methods give solutions to time dependent problems, but are often difficult to apply. A simulation model for dynamic system unavailability analysis can be developed which allows for easy construction and interpretation of complex dynamic reliability problems. Such a model could explicitly model interactions between components and include any desired capabilities such as various component repair states and testing and maintenance modeling features. There is a need for such a dynamic simulation model which can easily analyze complex systems and has the adaptability to be easily modified to handle a wide variety of problems (e.g., non-exponential transition times, dependent component failures, and control system reliability problems involving continuous process variables).

1.3 Organization of this Work

The purpose of this work is to propose a simulation model for dynamic system availability analysis. In Chapter 2 a survey is performed of the current techniques being employed in system reliability analysis. Their applications and limitations are addressed.

In Chapter 3 simulation languages are discussed briefly along with the specific characteristics of SIMSCRIPT II.5 which is being used for this simulation model. Program objectives are examined and all assumptions made are explained. The chapter concludes with a complete description of the

dynamic simulation model.

In Chapter 4 tests are performed on the simulation program and results are compared with selected established methods to demonstrate the program's validity. The procedures against which the model is compared include the GO-FLOW method and Markov chain techniques.

Chapter 5 presents a modification to the program to demonstrate the capability to model continuous variables. Specifically, the model is altered to perform the storage tank problem analyzed by Aldemir in ref. A-1 and ref. A-2. Results are compared with a Markovian analysis and the predictions of ref. A-1.

In Chapter 6 the results obtained with this model are summarized. The flexibility and adaptability of the simulation model are discussed along with limitations. The dynamic simulation model is compared with other reliability analysis techniques and their relative strengths and weaknesses cited. The chapter concludes with recommendations for future work.

Chapter 2

A Survey of Current System Reliability Analysis Techniques

2.1 Introduction

In this chapter a review is done of some of the current methods for reliability analysis. The literature has been surveyed and papers selected representing a cross section of the state of the art in reliability assessment. For ease of discussion these papers have been divided into two general areas under which all reliability analysis work can be categorized. These are static reliability analysis tools and dynamic system evaluation techniques.

In the following sections the current trends in both techniques are considered by reviewing recent literature. Figure 2.1 indicates the reliability analysis methods to be discussed and their categorization. Examples are used to illustrate unique features of the various procedures, and where appropriate, weaknesses in the methods are pointed out which could be avoided by using a dynamic simulation analysis approach. The chapter concludes with a summary section.

2.2 Static Methods

2.2.1 Fault Trees

One of the most familiar models used in system reliability analysis is the fault tree, described in reference W-1. The fault tree is a static system evaluation tool since it applies only to calculating the system reliability at a specified instant of time. To calculate dynamic information involves stepping forward in time and re-evaluating the tree.

Fault tree analysis is a deductive approach to system analysis and is used to compute the probability of an undesirable event, such as the non-

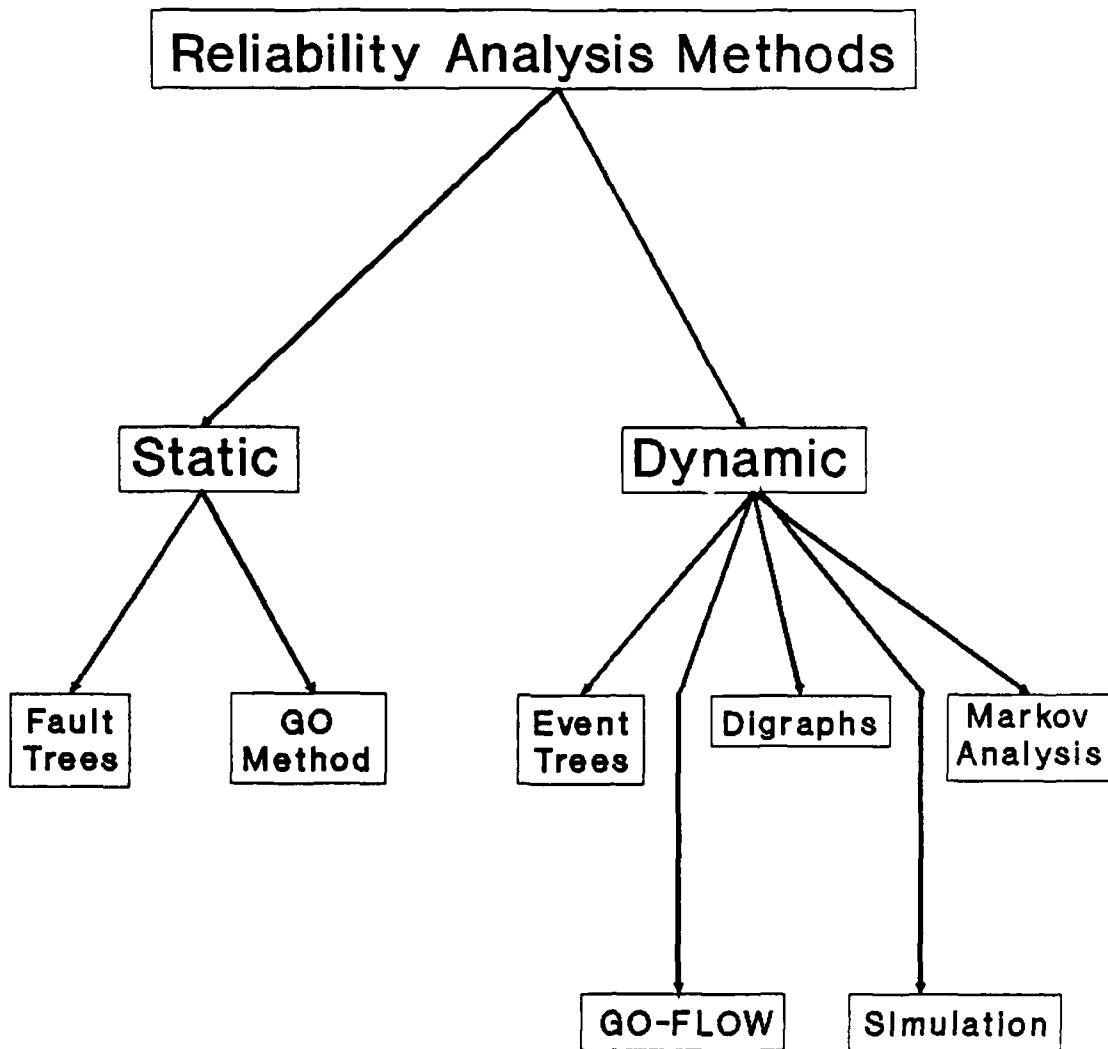


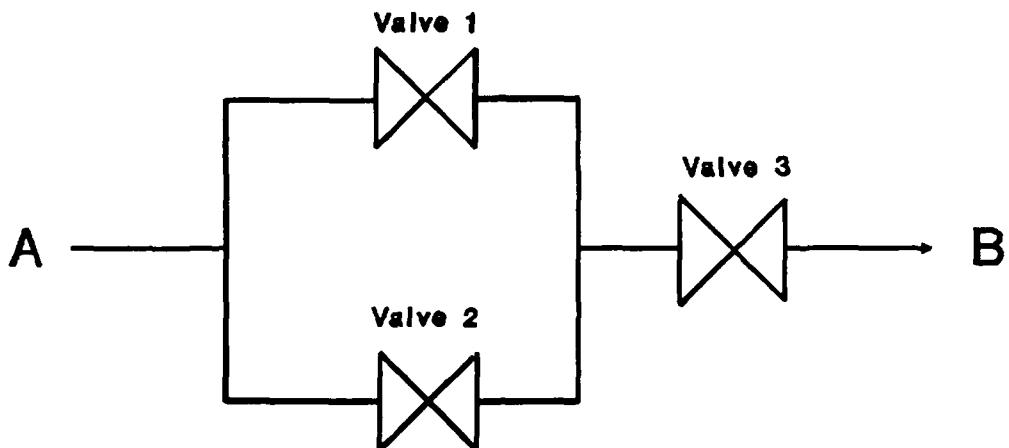
Figure 2.1 System Reliability Analysis Methods

operation of a system. The fault tree is a logic structure indicating how combinations of basic failure events can lead to the undesirable event of interest. The fault tree is then used to assess the probability that a system of components will be in a particular discrete state at a given point in time.

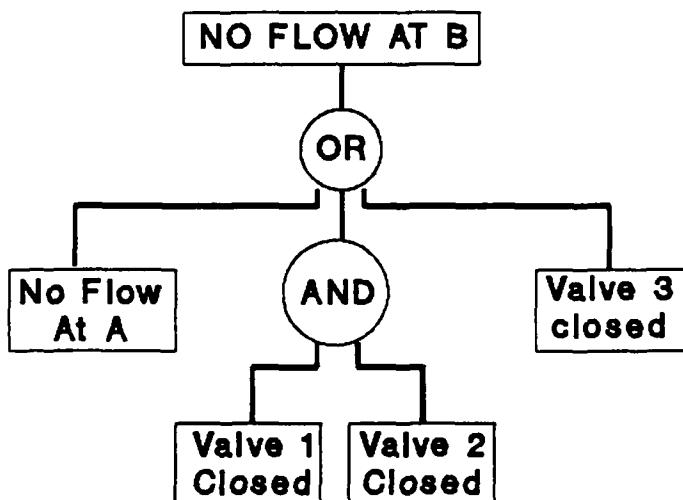
All fault tree analysis methods use a minimum of three logic operators. These are the AND gate, the OR gate, and the basic event or component, which has a set of failure data associated with it. Basic events in the tree refer to faults.

The methodology of fault tree construction consists of three steps. Step one is to identify the system to be analyzed and what boundaries are to be imposed. Step two is to determine the terminal failure event. This is the "top event" to be evaluated. And step three is to work backwards through the system to the component level to determine which combinations of component failures lead to system failure. This third step involves generating the logic structure known as the fault tree. Once the fault tree has been constructed, a boolean algebra expression can be written and solved numerically for the probability of the top event given the failure data concerning the individual components. Automated fault tree construction is possible using the CAT computer code as discussed by Apostolakis in reference A-4.

Consider the example shown in Figure 2.2A. The system consists of three valves which are supplied with flow from source A. Failure of the system is defined as occurring when flow is not present at point B. All three valves are normally open. In step one of constructing a fault tree for this system, the boundaries of the system are defined as the flow source A and the flow sink B. The second step is to determine the undesired event



a.) Diagram



b.) Fault Tree

$$P(\text{No Flow to B}) = P(\text{No Flow at A}) + P(\text{Valve 3 closed}) \\ + (P(\text{Valve 1 closed}) \cdot P(\text{Valve 2 closed}))$$

c.) Boolean Expression

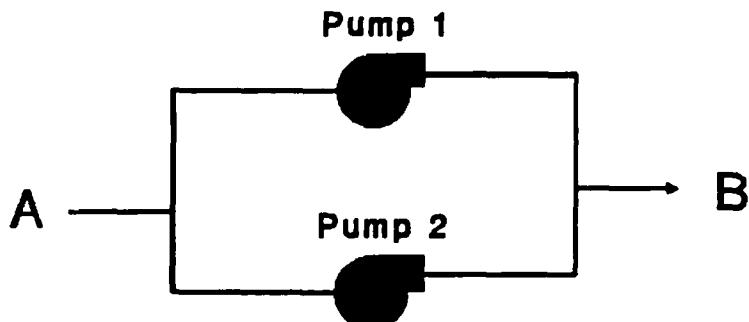
Figure 2.2 Fault Tree Example Problem

which for this example is loss of flow at point B. The third step is to identify which combinations of event can lead to the "top event." For this simple system, there will be no flow at B if valve three fails closed, if valves one and two fail closed, or if there is no flow from the source A. The fault tree can be constructed by tracing backwards from point B. Figure 2.2B shows the fault tree for the example.

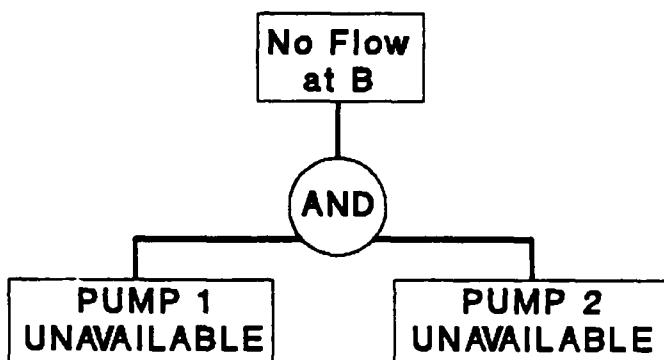
Based on the Boolean logic expressed by the fault tree, an expression can be written which quantifies the probability of the top event occurring. For the example, an approximate version of this expression is shown in Figure 2.2C. This expression can be evaluated numerically if the probabilities of all basic events are known. For complicated systems, computer codes are available which quantify the tree automatically.

Fault tree methods evaluate only the probability of a system being in a specified state at a given time, thus they do not directly apply to dynamic problems. But by thoughtful construction of a fault tree, and by evaluating the tree over and over again it is possible to treat dynamic problems in a discrete fashion. Repair and failure cycles can even be considered. Important limitations do exist, however. For instance, consider the example of Figure 2.3. The system is composed of two pumps providing flow to point A. Failure of the system occurs if flow is not provided by at least one of the two pumps, both of which are normally operating. If the failure rate of each pump depends on how many cycles of repair and failure it has been through, a fault tree of the system will be very clumsy (there will be a branch for each possible cycle). Thus a fault tree is not well suited for such a problem.

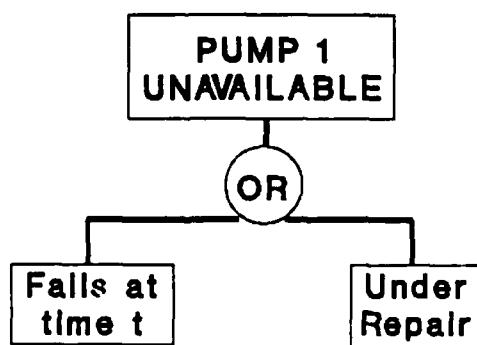
A variation on the fault tree method is described in reference D-2 by Dhillon and Rayapati. This method is similar to fault tree techniques but



a.) Diagram



b.) System Fault Tree



c.) Component Fault Tree

Figure 2.3 Fault Tree Limitation Example

involves systems composed of three state devices in series, parallel, series-parallel, and bridge configurations of any complexity. Any system which can be modeled in this fashion can be simplified through a series of reduction steps to a single equivalent three state component representing the entire system. This method assumes independence of all components.

Since a three state component is simply a model of a component which can fail open (on) or closed (off) and must be in one of these two failed states, or operational, it is clear that a fault tree can be constructed to model the same system and identical results would be obtained. The authors suggest this approach as possibly being of beneficial use to practical minded reliability engineers because of its simplicity and ease of use on appropriate problems.

2.2.2 GO Methodology

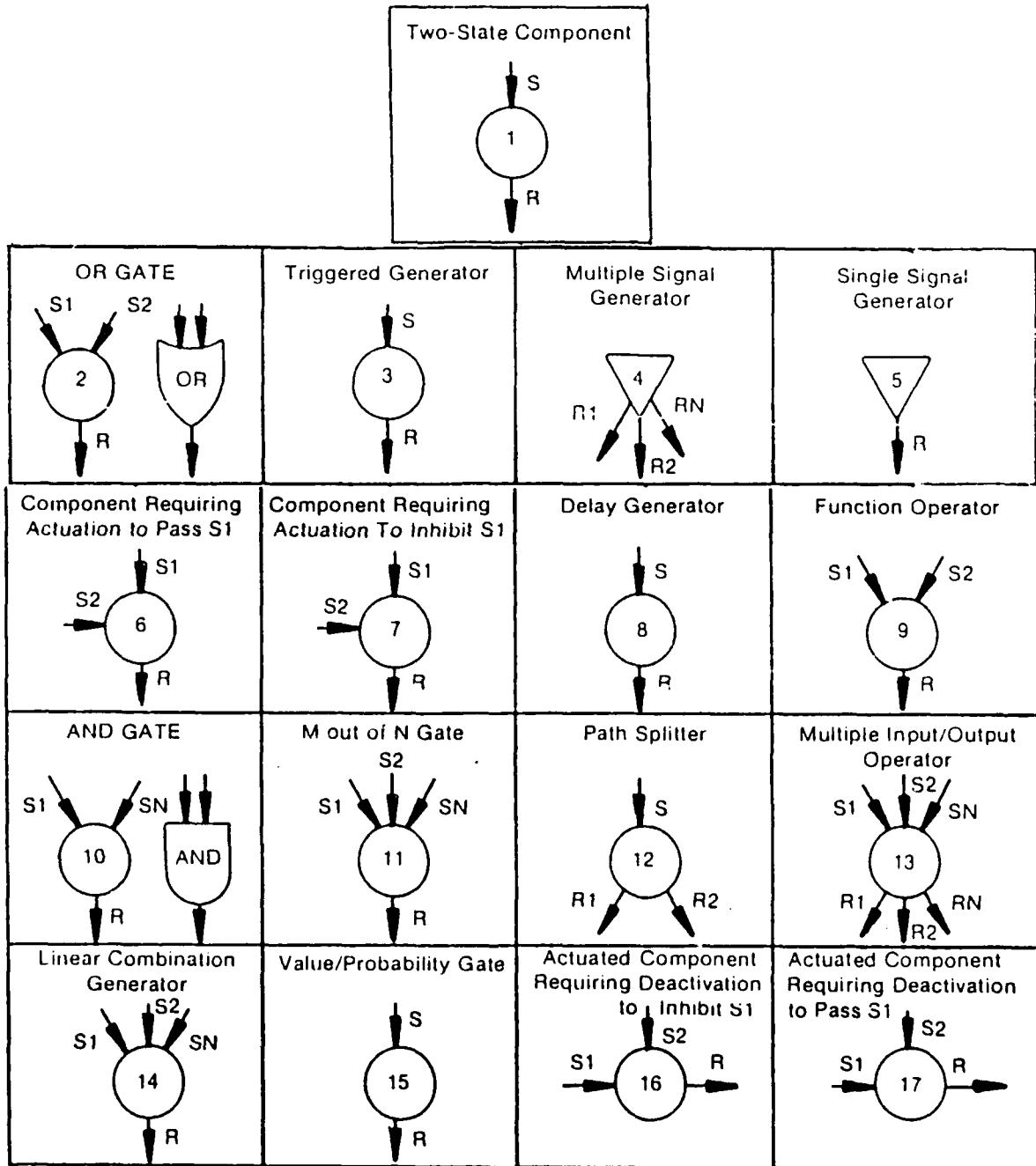
Another approach which solves for static system reliability is the GO methodology which was developed in the mid-1960's by Kaman Sciences Corporation. It gives results similar to fault tree evaluation computer codes, but differs significantly in its approach and structure. References G-1 and B-1 describe the GO procedures and a modified GO approach respectively.

The GO methodology is a "success oriented" technique which uses an inductive approach to determine the probability of a system being in a specified state. The procedure combines component probabilities and interactions to produce the probabilities of preselected output events. A set of standardized operators is available which are used to model all of the physical components in the system. Components are then linked using signals which represent the probability that the system is operational up to that point in the system diagram. This resulting GO chart in many cases

closely resembles a schematic diagram of the physical system layout.

There are seventeen basic operators in the GO method designed to incorporate the majority of logic functions found in physical system components. These operators include the three logic operators comprising the fault tree code methods plus fourteen additional ones which provide the capability to model the logic of most physical components with a single operator. Figure 2.4 shows the GO operators and is taken directly from reference G-1. For components which are not readily modeled by a simple operator it is possible to create a "supertype" which is a compilation of several operators to model a given component. This supertype can then be used to model the given component at every place it appears in the system. The object being to generate a GO chart model whose components very closely resemble the actual system components and operation.

The method of GO analysis consists of six basic steps. First, the system to be analyzed must be defined and all necessary information such as schematics, system description, success criteria, logic diagrams, and operating procedures must be gathered. Second, all inputs and outputs from the system, and each individual component must be determined and interrelated. A system may have many inputs and outputs unlike the fault tree method which has one output, the terminal event. To solve an identical problem using fault tree methods, several separate fault trees must be drawn. Third, a functional GO chart is drawn which indicates which components are dependent and which are independent operators. Fourth, each operator on the functional GO chart is assigned a type from one of the seventeen basic operators. Fifth, each operator is assigned a kind number so that if several valves in the system are identical, the failure data may be entered once for that "kind" of valve. Finally, the signal sequence must be identified so

Figure 2.4 GO Operators¹

¹ From Reference G-1 page 2-3.

that when the program runs the logical flow of the information in the model will be the same as the actual system.

The GO approach forces the analyst to begin the modelling process after defining system boundaries by inductively progressing from input to output by modelling the functional relationships between hardware components to get a successful input event to an output event. The basic events in the model are thus hardware components. Each component or operator, depending on its type, may exist in a variety of states, ie. open, closed, failed, failed prematurely, etc. This allows the model to more closely resemble actual system operation.

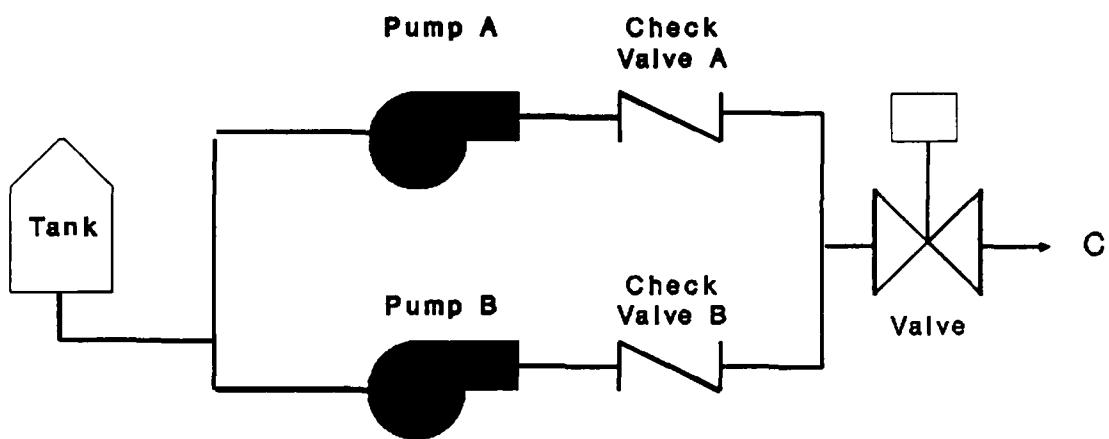
All signals existing in a GO analysis are numbers representing probabilities of existence of physical quantities, including flows, voltages, actuation signals, etc. Thus all signals have values between zero and one. When the GO program is run the logic operators operate on the signals in much the same fashion as the fault tree codes to produce the final output signal strength. The output or outputs will have values between zero and one indicating the probability of occurrence of the output event.

Figure 2.5A shows a schematic diagram of an example system containing a fluid source, two pumps, two check valves, and a remote operated valve. This system is modeled with GO operators in Figure 2.5B. The operators in the GO chart contain numbers indicating which type of GO operator is being used. The second number inside some of the operator symbols corresponds to the specific "kind" of operator. For example, operator number 6 is used to model both the two pumps and the motor operated valve. The two pumps are identical and are therefore assigned the same "kind" number, which in this case is 2. The valve is assigned a "kind" number of 4. When the GO program is executed, an input file is used which provides failure data for each

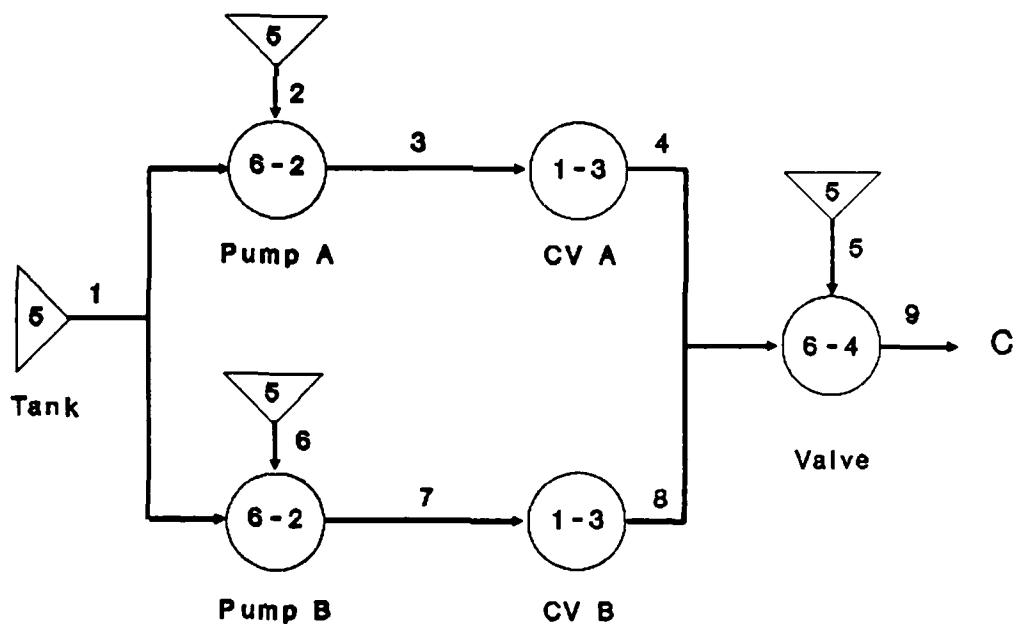
separate operator "kind".

It is readily seen from this example that the GO chart provides a system representation which very much resembles the physical plant layout. The model shows all components of the system and shows the relationships between them. The signals between components are numbered indicating the order in which system analysis progresses. Thus the GO method allows for time sequencing of events but does not provide dynamic availability information. Once a GO chart is created and an appropriate input file generated for computer evaluation, results are calculated in a manner very similar to fault tree analysis. The GO program will find and quantify all cutsets of the system, and print the minimal cutsets, if desired. The final result provided for the example of Figure 2.5 would indicate the probability of flow existing at point C at a specified instant of time. As with fault tree methods, to compute the system reliability at another time point will require a separate run of the program with a modified input file to reflect the new component failure probabilities.

The modified GO method proposed by Billinton and Patwardhan in reference B-1 presents simple changes to the original GO model. Since the basic GO method assumes independence of series elements, Billinton and Patwardhan point out that the method can lead to severe underestimation of the system reliability. The modification proposes using equivalent components to replace the individual components in a series system thereby incorporating the concept of dependence into series networks. The results are important for systems in which the failure of one series element does not prevent the subsequent failure of another element in the series while the first is under repair. Numerical examples illustrate the difference in results obtained from the two approaches. Both concepts are useful tools in



a.) System Diagram



b.) GO Chart

Figure 2.5 GO Example Problem

analyzing repairable systems.

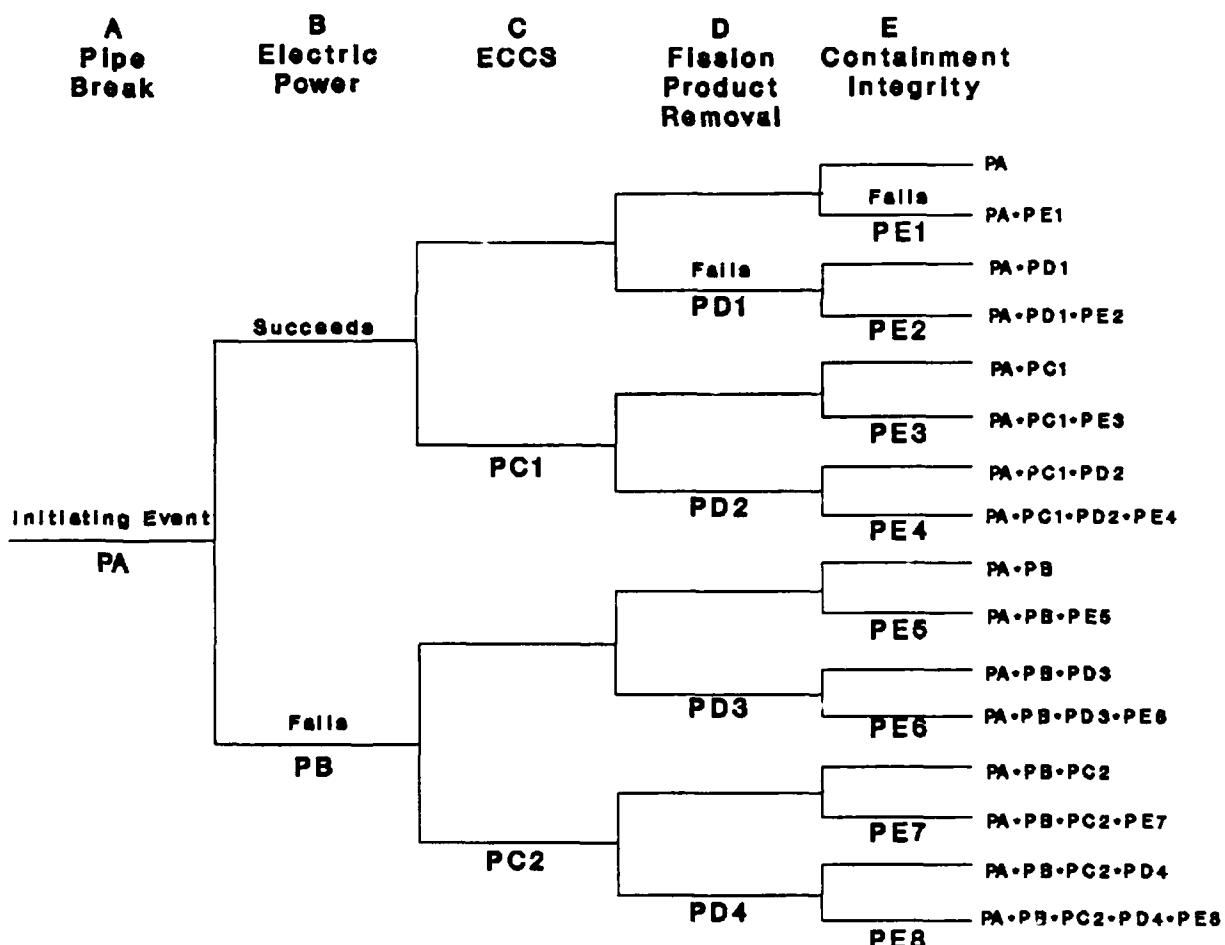
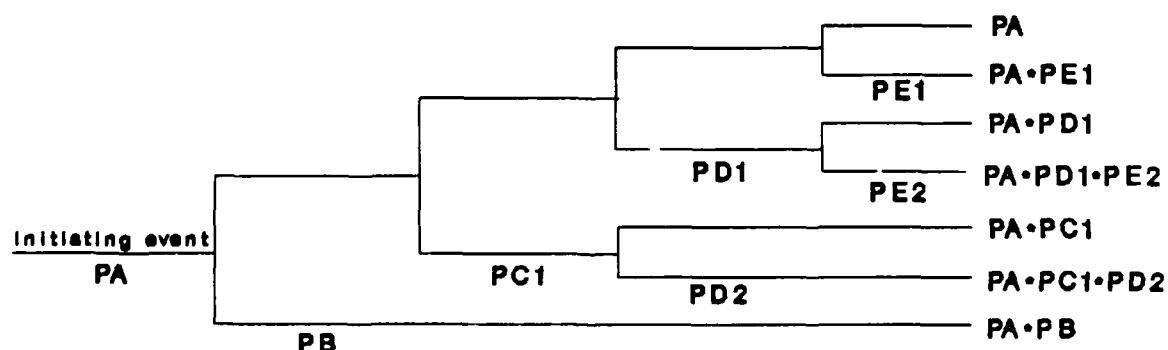
2.3 Dynamic Methods

2.3.1 Event Trees

Event trees are an inductive logic method used to identify the various possible outcomes of a given initiating event. The initiating event is normally some type of system component failure event or it could be an event external to the system. Event trees step through the possible consequences of the initiating event, and thus can be thought of as a quasi-dynamic reliability analysis tool since the passage of time is required between successive events in the event tree.

To illustrate the use of event trees, consider the example of Figure 2.6 which is taken from reference M-1. The example wishes to determine the probability of release or radioactivity from a nuclear reactor resulting from a large loss of coolant accident. The first step in event tree analysis is to identify the initiating event. For the example the initiating event is a large pipe break. The next step is to identify all applicable systems which may effect the outcome of the event. For the example, these are electric power availability, ECCS availability, fission product removal system availability, and containment integrity. The order in which these systems should appear in the event tree will depend on the logical relationship between them. Note that these logical relationships may conflict with the temporal sequence of events. For example, if the ECCS system depends partially on the availability of electricity, then electric power should come first in the event tree. In some problems, the order may be unimportant if systems are unrelated.

After all applicable systems are determined, then the success and failure states for each system must be determined. With this information,

**a.) Basic Event Tree****b.) Reduced Event Tree****Figure 2.6 Event Tree Example²**

² Taken from Reference M-1 page 195.

the tree of Figure 2.6A can be drawn representing all basic events. Some of the sequences shown may not be physically possible due to timing constraints and sequential and conditional dependencies, thus the next step is to eliminate the impossible states. The end result will be the reduced fault tree shown in Figure 2.6B. Each branch represent a possible outcome as a result of the initiating event. Each branch is quantified based on the transfer probabilities indicated. Determination of these individual transfer probabilities may involve the use of fault tree diagrams.

The example shows that event trees can be used in a limited sense to evaluate dynamic problems. The method requires that all system states be determined before quantification of the final state probabilities in much the same fashion as fault tree analysis. Once the event tree is identified and all transfer probabilities determined, calculation of terminal probabilities is straightforward and involves only simple multiplication as indicated in Figure 2.6. The major drawback of the event tree method is that it can not treat systems which contain loops. For example, in Figure 2.6, if the ECCS system fails and is then repaired it is desireable to transfer to another branch of the event tree. This can not be done without including a separate branch from the ECCS failed condition, which can certainly be done. However, if the problem to be analyzed contains an infinite possibility of component repair and failure cycles, the corresponding event tree must contain an infinite number of branches. This is not practical and thus limits the applicability of the event tree method to those systems which do not contain loops leading back to an event condition through which the analysis has already passed.

2.3.2 Digraphs

The method of digraphs is a tool used for analyzing relationships in

complex systems in which the desired operational state of one component may depend on the actual state of another component in the system or upon the level of a system process variable. These types of interactions are characteristic of all process control problems, and digraphs are a useful tool for analyzing causes of abnormal events in these systems. For example, a control system may monitor the level of fluid in a storage tank which supplies water to several different sources. Assume there are three pumps which can provide fluid input to the tank, and that the number of pumps operating at a given instant of time depends on the fluid level of the tank. If the tank is nearly empty, then all three pumps may be required to be on, while if the tank is almost full, only one, or none of the pumps may be required. A digraph analysis of this problem will treat the interaction of the fluid level causing pumps to turn on and off in a cause and effect type manner.

Reference K-1 by Kohda and Henley gives a detailed description of a digraph method. A digraph is a structure consisting of nodes and edges. Nodes represent process variables, or certain types of failure events and edges indicate a relation between two connected node variables. A digraph will resemble actual system configuration and can be easily constructed from a flowsheet of the process to be analyzed and a schematic diagram of the equipment. Fault trees are directly synthesized from digraphs and then analysis continues in the same manner as for fault tree methods. The additional capability is that the digraphs are designed to treat continuously variable processes. This procedure is ideally suited for process control type problems where component states depend on the value of a continuously varying signal, and where there is a need to determine the likelihood of deviating from steady state by a given amount. Results of the analysis

provide information on the probability that the system is in a given state at specified discrete time points. However, Kohda and Henley state in ref. K-1 that, in their view, none of the automated fault tree construction methods available, including the new one they propose, are successful in providing complete, consistent, and correct failure modes without human intervention and judgement. Thus the method of digraphs may involve considerable work to construct the digraph and then compose an appropriate fault tree.

Digraphs are a useful tool for identifying the minimal combinations of disturbance conditions which can lead to a specified undesirable condition at a digraph node. This is especially useful for optimizing control system design. The method includes dynamic system considerations, as the sequencing of component and process interactions must be considered when constructing the digraph. The analysis results obtained are not a dynamic representation of the system unavailability, but rather a listing of disturbances which can lead to undesirable performance of the system being analyzed. The method of digraphs requires complete knowledge of system behavior before analysis can begin, so that all possible system disturbance modes are identified and included in the digraph analysis.

2.3.3 GO-FLOW Methodology

Developed by Matsuoka and Kobayashi, the GO-FLOW method was generated in the mid-1980's as a success-oriented system analysis technique to analyze the time dependent unavailability of phased mission problems. References M-2 and M-3 give a detailed description of the method, philosophy, and structure as well as several illustrative examples.

The GO-FLOW methodology is derived from the GO code and is therefore similar to it in many respects. The GO-FLOW method uses an inductive

approach and a combination of ten basic operators to create a GO-FLOW chart which very much resembles a schematic diagram of the system being modeled. The GO-FLOW operators are shown in Figure 2.7, which is taken from reference M-2. Table 2.1, also taken from reference M-2 summarizes the output relationships for each of the operator types. The idea is to create a reliability model which can be easily understood by the design engineer. The modeling process for GO-FLOW is much the same as for the GO methodology.

The GO-FLOW chart will look similar to a GO chart except for the fact that there are fewer basic operators available in the GO-FLOW method. All GO-FLOW operators have corresponding operators in the GO code but the operation of each is slightly different. Each of the seven additional operators used by the GO code can be modeled using combinations of the other basic operators in the GO-FLOW code.

The major difference between the two methods is the manner in which signals between operators are treated. The signals in a GO model correspond to the probability that a component is in a given state independent of time. Thus, a time dependent system analysis cannot be performed. Signals can go from "on" to "off" or vice versa but they can not go from "on" to "off" and then back to "on." Thus it is not directly possible to incorporate repair in a GO model. In a GO-FLOW model, on the other hand, many signals represent probabilities, but they are functions of discrete time points. Some signals represent only time information to cause proper activation of time dependent operators at the proper instant during the process of the analysis. Thus a GO-FLOW model, unlike the GO model, is time dependent. The GO model incorporates time sequencing in its analysis but the GO-FLOW method simulates the actual passage of time through discrete time steps. Signals in a GO model indicate a change of state or an occurrence whereas in the GO-FLOW

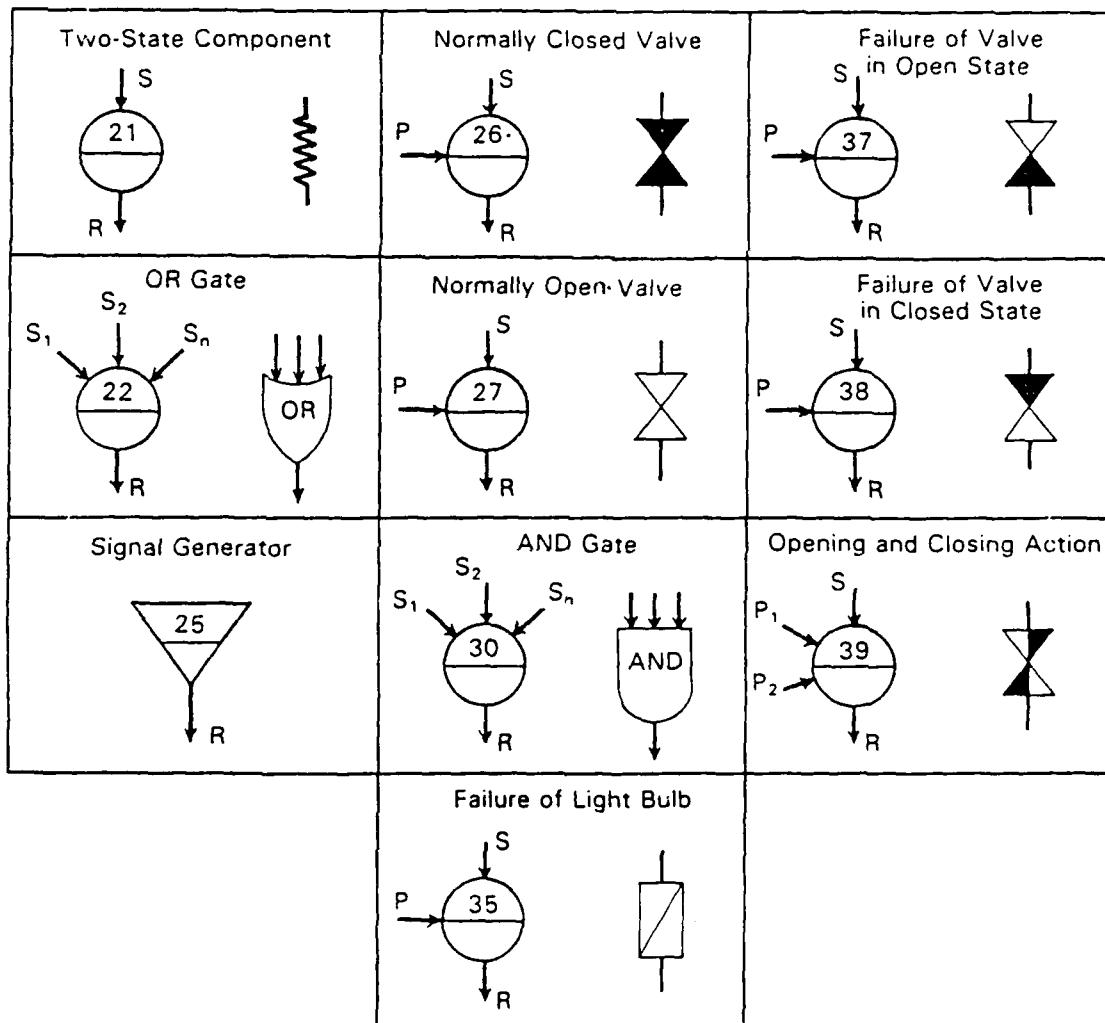


Figure 2.7 GO-FLOW Operators³

³ Taken from Reference M-2 page 66.

Table 2.1
Summary of the Function of GO-FLOW Operators⁴

| Operator Type | Main Input Signal Intensity | Subinput Signal Intensity | Output Signal Intensity |
|---------------|---------------------------------|---|--|
| 21 | $S(t)$ | --- | $R(t) = S(t) \cdot P_t$ |
| 22 | $S_1(t), S_2(t), \dots, S_n(t)$ | --- | Probability that at least one input signal exists |
| 25 | --- | --- | Probability of a demand or time duration |
| 26 | $S(t)$ | $P(t)$ | $R(t) = S(t) \cdot O(t), O(t_1) = P_p,$ $O(t) = O(t') + [1.0 - O(t')] \cdot P(t) \cdot P_t$ |
| 27 | $S(t)$ | $P(t)$ | $R(t) = S(t) \cdot O(t), O(t_1) = 1.0 - P_p,$ $O(t) = O(t') \cdot [1.0 - P(t) \cdot P_t]$ |
| 30 | $S_1(t), S_2(t), \dots, S_n(t)$ | --- | Probability that all the input signals exist |
| 35 | $S(t_1), S(t_2), \dots, S(t)$ | $P_1(t_1), \dots, P_1(t_n)$ $P_2(t_1), \dots, P_2(t_n)$ \dots | $R(t) = S(t) \cdot \exp \left\{ -\lambda \sum_{t_i \leq t} \sum_{t_k \leq t_i} P_i(t_k) \min \{1.0, S(t_k)/S(t)\} \right\}$ |
| 37 | $S(t)$ | $P_1(t_1), \dots, P_1(t_n)$ $P_2(t_1), \dots, P_2(t_n)$ \dots | $R(t) = S(t) \cdot \exp \left[-\lambda \sum_{t_i \leq t} \sum_{t_k \leq t_i} P_i(t_k) \right]$ |
| 38 | $S(t)$ | $P_1(t_1), \dots, P_1(t_n)$ $P_2(t_1), \dots, P_2(t_n)$ \dots | $R(t) = S(t) \cdot \left\{ 1.0 - \exp \left[-\lambda \sum_{t_i \leq t} \sum_{t_k \leq t_i} P_i(t_k) \right] \right\}$ |
| 39 | $S(t)$ | $P_1(t)$ $P_2(t)$ | $R(t) = S(t) \cdot O(t),$ $O(t) = O(t') + [1.0 - O(t')] \cdot P_1(t) \cdot P_p,$ $R(t) = S(t) \cdot O(t), O(t) = O(t') \cdot [1.0 - P_2(t) \cdot P_t]$ |

⁴ Taken from Reference M-2 page 67.

model signals represent time dependent component state information. This is the major point of difference between GO-FLOW and its predecessor.

With this time capability GO-FLOW has the additional potential of being able to perform an unavailability analysis incorporating repair in the model. To do a time dependent analysis with the GO code would require several runs of the program and changing the input information for each separate run. The GO method can find the time when a system changes from one state to another but it can not analyze a system which has more than one state change. In the GO-FLOW methodology signals correspond to the existence of physical quantities at various points in time. Sub-input signals to certain operators represent time information rather than probabilities and therefore can take on values greater than one. Signals are time dependent reflecting the fact that the GO-FLOW code is set up to perform time dependent unavailability analysis.

Figure 2.8 shows the GO-FLOW chart for the light bulb problem analyzed in section five of Chapter 4. This figure is taken from reference M-2. The actual system diagram is shown in Figure 4.10 of Chapter 4. Each operator in the GO-FLOW chart of Figure 2.8 contains an operator type number corresponding to one of the 10 basic operators. This is the top number in each symbol. The lower number is simply the component number assigned. Each operator in the system must have a separate component number. Component number 4 is shown twice in the figure because it supplies a signal to two separate components. All signals in the system are also numbered sequentially indicating the time sequence which will be used in system analysis. Table 2.2 summarizes the operators used in Figure 2.8. Table 2.3 defines the signals of the example system. Both of these tables are from reference M-2.

Using the equations associated with each operator, as shown in Table

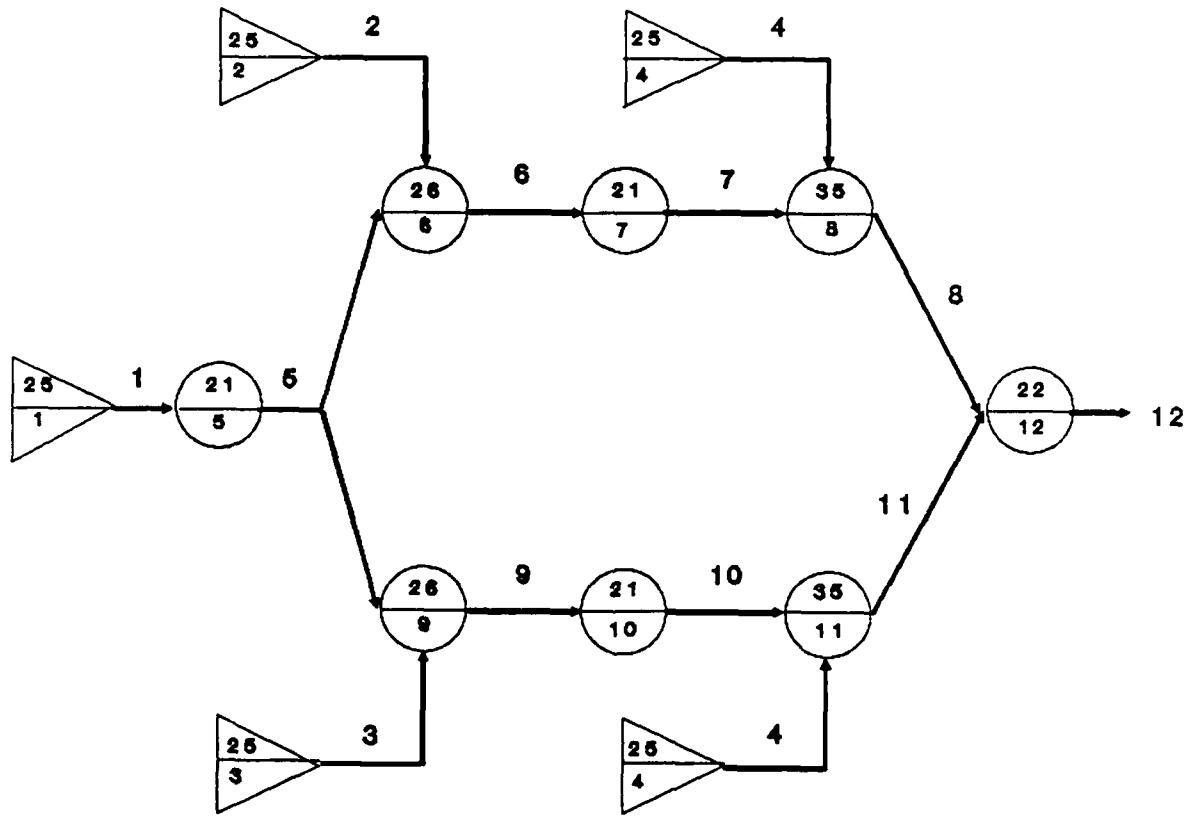


Figure 2.8 GO-FLOW Chart Example

2.1, it is possible to step through the analysis and determine the probability that at least one light is lit at each of the designated time points. Specifically, time points 1, 2 and 3 correspond to time $t=0.0$. Time point 1 is prior to any system change, time point 2 corresponds to connecting the battery, and time point 3 is when switch S1 is closed. Time point 4 corresponds to time $t=10.0$ hours, but prior to closing of switch S2, and time point 5 corresponds to switch S2 being closed (at $t=10.0$ hours). Time point 6 is at $t=20.0$ hours. The output signal of each operator is determined at each time point by applying the operator equations. Signals are evaluated sequentially as numbered since input signals to operators must be determined before outputs can be calculated. Time points are also evaluated sequentially after all signals for the preceding time point are calculated, because certain operator types require knowledge of the previous input and output signals. These calculations are all done internally by the GO-FLOW program, but can be done manually if desired.

For the example problem shown if Figure 2.8, which is also treated in Chapter 4, manual calculations were performed to calculate output signal intensities at each of the designated time points. The results indicate the probability that the indicated signal exists at the selected time point. These results are compared with simulation estimates in Chapter 4. Table 2.4 provides a summary of the signal strengths calculated at each of the time points. The results shown for signal 12, the output of operator 12, are used in Chapter 4 for comparison with the results obtained using the simulation program method.

Table 2.2
Operators Used in GO-FLOW Sample Problem

| <u>Operator</u> | | <u>Type</u> | <u>Number</u> | <u>Data</u> | <u>Meaning</u> |
|-----------------|----|-------------|---------------|--|--|
| 25 | 1 | | | R(1)=0.0, R(t)=1.0 (t not equal 1) | Battery is connected |
| 25 | 2 | | | R(3)=1.0, R(t)=0.0 (t not equal 3) | Demand signal for switch S1 to close |
| 25 | 3 | | | R(5)=1.0, R(t)=0.0 (t not equal 5) | Demand signal for switch S2 to close |
| 25 | 4 | | | R(4)=10.0,R(6)=10.0, R(t)=0.0 (t not equal 4,6) | Time duration |
| 21 | 5 | | | P _s = 0.9 | Battery works with probability 0.9 |
| 26 | 6 | | | P _s = 0.7 | Switch S1 closes normally with probability 0.7 |
| 21 | 7 | | | P _s = 0.8 | Light bulb L1 works with probability 0.8 |
| 35 | 8 | | | λ = 0.001/h | Failure of L1 while on |
| 26 | 9 | | | P _s = 0.7 | Switch S2 closes normally with probability 0.7 |
| 21 | 10 | | | P _s = 0.8 | Light bulb L2 works with probability 0.8 |
| 35 | 11 | | | λ = 0.001/h | Failure of L2 while on |
| 22 | 12 | | | | OR gate |

Table 2.3
signals Defined in Sample GO-FLOW Problem

| <u>Signal</u> | <u>Definition</u> |
|---------------|--|
| 1 | Battery is connected |
| 2 | Demand signal for S1 to close |
| 3 | Demand signal for S2 to close |
| 4 | Time duration |
| 5 | Battery provides power for both lights |
| 6 | Power is available at L1 |
| 7 | L1 lights up without demand failure |
| 8 | L1 is on |
| 9 | Power is available at L2 |
| 10 | L2 lights up without demand failure |
| 11 | L2 is on |
| 12 | Either L1 or L2 is on (final signal) |

Table 2.4
Calculation Steps for Sample Problem

| Operator Number | Signal | | Intensity of Output Signal for Time Points 1 to 6 | | | | | |
|--------------------|--------|--------|---|-----|-------|--------|--------|--------|
| | Input | Output | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | --- | 1 | 0.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| 2 | --- | 2 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 |
| 3 | --- | 3 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| 4 | --- | 4 | 0.0 | 0.0 | 0.0 | 10.0 | 0.0 | 0.0 |
| 5 | 1 | 5 | 0.0 | 0.9 | 0.9 | 0.9 | 0.9 | 0.9 |
| 6 | 5,(2) | 6,(5) | 0.0 | 0.0 | 0.63 | 0.63 | 0.63 | 0.63 |
| 7 | 6 | 7,(5) | 0.0 | 0.0 | 0.504 | 0.504 | 0.504 | 0.504 |
| 8 | 7,(4) | 8,(5) | 0.0 | 0.0 | 0.504 | 0.4990 | 0.4990 | 0.4940 |
| 9 | 5,(3) | 9,(5) | 0.0 | 0.0 | 0.0 | 0.0 | 0.63 | 0.63 |
| 10 | 9 | 10,(5) | 0.0 | 0.0 | 0.0 | 0.0 | 0.504 | 0.504 |
| 11 | 10,(4) | 11,(5) | 0.0 | 0.0 | 0.0 | 0.0 | 0.504 | 0.4990 |
| 12 | 8,11 | 12 | 0.0 | 0.0 | 0.504 | 0.4990 | 0.7236 | 0.7191 |

2.3 Markovian Analysis

Markov models are useful in determining steady state and time dependent availability measures. Markov analysis techniques are predominantly applied to systems which are normally solved in continuous time. The major drawbacks of the method are that it can not easily be used to treat phased mission problems and the basic Markov models require all state transition rates to be exponentially distributed.

To solve a complex phased mission problem using the Markov method would require using Markov chains in which the probability of being in any state is calculated up to the point where the possible states of the system are changed using straightforward Markov analysis. Then the new states are added to the analysis and new Markov equations are written. The results from the first stage are used to define the initial probabilities of being in any of the new states. The analysis progresses in time until the next point where the state space of the system is changed by an external event such as maintenance, testing, or a control function. The chain method can be used

to analyze process control system problems by discretization techniques, however for complex problems the Markov chains may become unmanageably large.

Many fundamental system reliability and risk analysis texts such as references B-2, D-1, M-1, and P-1 give adequate descriptions of the basic Markov modeling techniques. The essential step is first to identify all of the possible states the system can be in. For a system of twenty components which can be in either an operating or a failed state there are 2^{20} possible system states indicating the need for reduction techniques to simplify the problem. These methods often involve the use of absorbing states and simple numerical procedures allowing for truncation and approximation of values.

Reference P-3 by Papazoglou and Gyftopoulos discusses the technique of merging which can also be used to reduce the order of a Markov system. This, of course, can lead to results which are not exact, but can be made to have arbitrarily small error percentages.

Once all system states are identified, a set of first order differential equations is written describing the rate of transfer in and out of each state. If the transfer rates are exponential and are constant with time, then the system is described by a set of linear, constant coefficient, first order differential equations and can easily be solved by any one of a number of methods. These include matrix exponentiation, Laplace transforms, eigenvalues, and discrete time integration techniques such as the Runge-Kutta method. For cases where state transition times are not exponential, the system of differential equations are no longer Markovian. These non-Markovian systems, which can be represented by discrete state spaces, may be solvable by one of the three techniques described in ref. P-1. These include the method of supplementary variables, the dummy states method, and the embedded chain approach. All of these techniques involve attempts to reduce

the system to an equivalent Markovian problem. However, large systems which can be explicitly modeled by these techniques are very rare and thus without the use of simplifying assumptions or approximate mathematical procedures, the use of these methods is extremely limited.

Current work in the area of Markovian analysis centers around two areas. These include methods to reduce the order of the system and methods to incorporate dynamic system characteristics for modeling systems with phased mission scenarios. Several papers on recent work are considered here.

In reference J-1, Johnson discusses a general availability model and basic modeling techniques. Rules for writing the state transition matrix and its eigenvalues and eigenvectors are given. Then formulas are discussed for computing approximate results for state probabilities based on the eigenvalues and eigenvectors. Three fundamental cases are examined to demonstrate the approach for determining steady state availability values. These three cases are a system involving no repair, a system of N components having identical repair and failure rates, and an N component system in which only one component can be failed at a time. The paper covers basic concepts and gives a general review of Markov modeling.

Jeong, Chang, and Kim propose applying Markovian analysis techniques to fault tree evaluation in reference J-2. The purpose is to produce time dependent availability results of a continuous nature directly from the fault tree structures and to correctly incorporate component maintenance and testing, i.e. to model dependencies on the state of the system. Their method is to describe basic events in a modified fault tree by Markovian analysis techniques. The new tree contains numerous Markov states and includes system state dependencies such as whether or not components are undergoing maintenance. This new discrete state, continuous time model is quite large,

thus Jeong, Chang, and Kim introduce the concept of a super component to reduce the order of the transition matrix. These super components contain a number of basic events from the fault tree and provide the fundamental means by which the tree is converted to a Markovian process with a minimal number of states to be solved for. Results provide dynamic availability data for the system while incorporating maintenance and testing.

Reference G-2 by Gray deals with simplifying complex systems. The method applies only to systems containing parallel active and redundant subgroups of identical components and should prove useful in designing and analyzing redundant systems. Each subgroup is analyzed using Markov techniques to determine the time dependent behavior of all components in the subgroup within the context of the subgroup as a separate unit. The reliability of the subgroup can then be determined. From the expression governing this relationship it is possible to calculate a mean and a standard deviation of the subgroup time to failure. If the time to failure can be modeled as exponentially distributed then the subgroup can be replaced by an "equivalent element." Then further decomposition may continue if a parallel subgroup of "equivalent elements" can be modeled in the same fashion.

This method can greatly reduce the order of a complex system to be solved by Markovian analysis and therefore reduce the computer time necessary relative to matrix powering techniques used to solve the same problem. However, application is limited, as the method requires that there be parallel subgroups of like components, and even when such structures exist, they can only be replaced with "equivalent elements" if analysis of the subgroup shows it to have exponential repair and failure distributions, which certainly may not always be the case.

An important characteristic of the Markov model is that at any time the

system can be completely described by specifying the state the system is in and all of the exponential transfer rates in and out of each system state. Solving the system of equations provides time dependent information about the probability that the system is in any one of the possible states, however if it is necessary to know the probability of transition between any two states at a specified time it is essential to use the Chapman-Kolmogorov equations.

The Chapman-Kolmogorov equations are given by:

$$P_{jk}(t,t') = \sum_{i=1}^N P_{ji}(t,u) P_{ik}(u,t') \quad (2.1)$$

where $t' \leq u \leq t$

These allow direct calculation of transition probabilities provided all possible intermediate transitions are determined first. In reference L-1 Limnios describes a new method for numerical solution of the Kolmogorov equations for continuous time Markov chains. The method provides a very simple numerical treatment which can be applied directly i.e. discrete time solutions of Markov systems. The error can be made arbitrarily small through proper choice of parameters. The procedure requires less operations than direct development for cases where the norm of the state transition matrix is large (greater than five). The larger the norm of the transition matrix, the greater the computational savings. This method will prove useful in numerical computer codes which solve Markov systems through the use of matrix exponentiation techniques.

Another variation of the Markov approach is discussed by Aldemir in references A-1 and A-2. This method describes a dynamic failure model for evaluation of process control systems. The approach is unique in that most techniques consider simple binary components which are either failed or operational and this method considers continuous state dynamic variables

including such parameters as temperature, pressure, and liquid level. The method, which is described in detail in reference A-1, is based on discrete state space, discrete time representation of process control system dynamics and probabilistic system behavior simulated by Markov chains. An algorithm is developed for construction of the state transition matrix and input preparation for the algorithm is illustrated by example. The method is demonstrated to be effective for accurate failure analysis of process control systems.

Markovian analysis procedures provide a means for determining system state continuous time (or discrete time) availability information. The method applies only to components with exponentially distributed repair and failure rates. Once the state transition equations are derived it is always possible to solve the system exactly although in many cases exact solution may require a prohibitive amount of computational effort. The mathematics involved can become extremely cumbersome as the size of the system increases since the number of system states expands exponentially with the number of system components. The method also does not lend itself well to phased mission analysis although Markov chains can be used as discussed above. Example Markov solutions are shown in conjunction with the sample simulation problems considered in Chapters 4 and 5.

2.3.5 Monte Carlo Simulation

Monte Carlo simulation is a system reliability analysis approach which has great potential for solving complex analysis problems. Descriptions of the method of Monte Carlo simulation for system reliability analysis can be found in references B-2 and P-1 and many other systems reliability analysis texts. The basic approach involves generating a computer model which simulates operation of the system under consideration. Random number

generators are used to model the random events in the system such as failure and repair of components. For each set of random numbers (experiment), the program is run and the result is a discrete time description of all system process variables and/or system component states. The experiment is performed a series of times and the results are averaged over all of the trials to provide an estimate of the system reliability. For example, if the availability of a system, $A(t)$, is to be determined, the simulation can be run for N trials, each simulating system operation over a fixed time period, T . The value of $A(t)$ for each trial can be stored for selected discrete time points to provide $A(t | \text{trial } i)$. An estimate of $A(t)$ can then be made at the discrete time points using the equation:

$$A'(t) = \left[\frac{1}{N} \right] \sum_{i=1}^N A(t | \text{trial } i) \quad (2.2)$$

where $A'(t)$ is an estimate of $A(t)$

As N goes to infinity this estimate will approach the exact analytical result. The variance of the estimate $A'(t)$ is proportional to $1/N$, thus to reduce the standard deviation of the estimate it is necessary to increase the number of trials performed.

There are advantages and disadvantages to the use of Monte Carlo simulation. One major advantage is that since a simulation model is developed to fit the problem, it is possible to quantitatively evaluate any system regardless of the form of the transfer rate expressions and the complexity of the system itself. In theory, results can be made as accurate as desirable by improving the modeling assumptions and running the experiment for an arbitrarily large number of trials. However, herein also lies the major disadvantage. To develop extremely accurate models for complex systems may involve a relatively large amount of time for formation

of the program and excessive cost in computer time for execution since the number of trials required to obtain acceptable results may be unreasonable. And, in general, references G-3 and P-1 agree that the amount of simulation time required to provide reasonable confidence in the results obtained will be far larger than the time required to acquire an analytical solution to the same problem, provided an analytic solution can be achieved. Monte Carlo simulation methods are being used in many engineering fields and one example is in the area of electric power generating system reliability. References A-3 and G-3 both deal with this subject in detail. In reference G-3, Ghajar and Billinton use Monte Carlo simulation to produce a generating system outage history over a period of time combined with a load model to determine adequacy indices. In this model start-up and shut-down of individual generating units are modeled and start-up failures are modeled distinctly from running failures. The model is applied to the IEEE Reliability Test System (RTS) and it demonstrates good agreement with analytical results.

Reference A-3 by Allan, Jebril, Saboury, and Roman, similarly evaluates power system reliability using Monte Carlo simulation, but with a slightly different model. Results of this model for the IEEE RTS are compared with the analytical results and again the mean values of the indices indicate favorable outcomes. It is noted, however, that the standard deviation or dispersion of the results is extreme even though mean values are consistent. This can be of major importance since relevant decisions may be made based on dispersion characteristics of certain indices. For this model it may be necessary to run a much larger number of trials or possibly to employ variance reduction techniques.

Reference L-2 by Lewis and Zhuguo describes how Monte Carlo sampling procedures are used to solve inhomogeneous Markov processes. Inhomogeneous

Markov processes refer to systems which can be modeled by a discrete state space, but the transition rates are time dependent. These problems are solved with Markovian analysis techniques by using Monte Carlo sampling to determine the times between state transitions. It is explained that Monte Carlo methods are capable of treating Markov models which would be intractable by deterministic computational methods. In the reference, a model is developed which can treat Markov systems with time dependent transition rates and which allows for periodic testing and maintenance. The paper concludes with remarks concerning the development of future Monte Carlo simulation models for system reliability analysis which incorporate the concept of cumulative damage.

Kumamoto, Tanaka, and Inoue describe a new Monte Carlo method for evaluating system failure probabilities in reference K-2. They explain that for extremely reliable systems the method of direct Monte Carlo simulation is not applicable since the number of trials required to obtain meaningful results would be prohibitive. This fact is well known and variance reduction techniques are applied to yield smaller variances with the same number of trials as direct Monte Carlo methods. Karp and Luby have previously developed the Karp-Luby minimum variance estimator (KLM) which is used to estimate the minimum number of trials necessary to achieve results which are accurate within certain variance limits. The KLM method can only be applied to a system if all the minimal cut sets are known. A variance is specified and then the KLM estimates the number of trials necessary to achieve satisfactory results with the Karp-Luby Monte Carlo method.

Kumamoto, Tanaka, and Inoue have developed what they call the "New Coverage Monte Carlo" (NCM) method and an associated minimum variance estimator. Their method is very similar in nature to the Karp-Luby method,

however they are able to achieve results of equal accuracy with a far fewer number of trials thus saving considerable computer time. Their method can only be applied to systems composed of components with very small failure probabilities and the smaller the component failure probabilities are the more pronounced the benefit this method provides over the Karp-Luby method.

Monte Carlo simulation methods are seen to be extremely useful in evaluating system reliability for complex systems which often can not be solved analytically. This method also allows for straightforward analysis of phased mission problems. It can be, however, quite time consuming to use and the results can have large variances associated with them.

2.6 Chapter Summary

The many methods for system reliability analysis can be broadly grouped into two basic categories. Those methods which provide static system information and those which can be used to do dynamic analyses. Of the static methods, fault tree analysis procedures and the GO methodology were considered. For dynamic analysis methods, event trees, digraphs, Markovian analysis, the GO-FLOW methodology, and simulation were discussed.

The two static methods provide excellent tools for evaluating average system unavailability but have only limited applicability to solving dynamic problems. Separate program runs or computation sets must be done to evaluate the system at each discrete time point of interest. Also because of their static nature, they can not be used to solve phased mission problems without performing numerous intermediate analysis computations.

Of the dynamic methods it was seen that availability information can be obtained for all manner of complex systems, however each method has its limitations or drawbacks. Event trees can not be used to treat systems which contain an indefinite number of failure and repair loops and are only quasi-

dynamic, as they involve the passage of time in the event tree sequencing of events. Markovian analysis provides detailed information about time dependent unavailability, but can only be applied to systems containing components with exponentially distributed transition times. GO-FLOW allows for consideration of any manner of transition time distributions, however it is only designed to treat discrete state devices. If continuously varying process variables are to be considered a complex discretization scheme must be developed.

With the possible exception of the GO and GO-FLOW methodologies, all system reliability analysis techniques discussed require development of a model where all possible system states are clearly defined. An alternative to this type of problem analysis is the simulation approach. A Monte Carlo simulation approach can be developed which requires only an understanding of the components involved in the system and the relationships between them. In the next chapter a benchmark model is developed which provides a simple approach to determining the availability of complex systems.

Chapter 3

DYMCAM Dynamic Simulation Model

3.1 Introduction

In Chapter 2, one of the methods discussed for evaluating the reliability of complex systems was the Monte Carlo simulation technique. There are many ways that this technique can be implemented using a computer, and the way the computer program functions, along with the usefulness of the simulation results, can be dependent on the simulation language which is used. In this chapter simulation languages are discussed in general and their benefits and weaknesses with regard to Monte Carlo simulation programming are examined. The SIMSCRIPT II.5 simulation language is used for the dynamic simulation model developed in this work. It is described in the second section of this chapter along with an explanation of why this language is believed to be useful for a Monte Carlo simulation model of the type developed in this work. The SIMSCRIPT language is a powerful simulation tool and has not been used to its fullest extent in this work. The program developed here is a basic model designed to demonstrate fundamental features of a simulation approach to availability analysis, and limitations of this program should not be viewed as limitations of the SIMSCRIPT language or simulation in general.

Following the discussion of simulation languages, the subsequent sections of this chapter describe in detail the design objectives that were used in developing the DYMCAM (DYnamic Monte Carlo Availability Model) program proposed. Certain assumptions were made to keep the initial DYMCAM program at a manageable size for this introductory work, and these assumptions are detailed explicitly in Section 3.3. The fourth section gives a detailed description of the DYMCAM program. This program has been written

in a very general form and throughout this work it is pointed out where simple modifications can be made to the program to make it more powerful or to meet specific needs. Many such modification proposals are included in the program description section of this chapter. For a detailed description of the input file format necessary to run the program, Appendix A should be consulted. The chapter concludes with a summary section.

3.2 Simulation Languages

3.2.1 An Overview

To fully appreciate the method of Monte Carlo simulation for systems reliability analysis, it is necessary to understand simulation as a tool, and what its advantages and disadvantages are. Reference B-4 by Banks and Carson is an excellent text for studying simulation techniques. In addition, in reference B-5, also by Banks and Carson, there is a good description of several process-interaction simulation languages. Different simulation languages are designed to be used for certain types of problems and it is necessary to know which languages are best used for Monte Carlo problems. This section takes a general look at simulation approaches.

It should be pointed out that in the area of system reliability analysis, simulation can be an important tool since there are numerous examples of large complex problems which cannot be solved by analytical techniques. Monte Carlo procedures are very useful for many such instances. Further, even in cases where analytical solutions are available it may be desirable to use simulation methods. The advantages and disadvantages must be understood, however. Presently, the methods of simulation are not widely used in reliability analysis. Reference S-1 by Schmidt and Taylor specifies some primary advantages and disadvantages of simulation and these are listed in Table 3.1.

Table 3.1**Advantages and Disadvantages of Simulation****Advantages:**

- Once a model is built, it can be used repeatedly to analyze proposed designs or policies.
- Simulation models are usually easier to apply than analytic methods. Thus there are many more potential users of simulation models than of analytic methods.
- Whereas analytic models usually require many simplifying assumptions to make them mathematically tractable, simulation models have no such restrictions. With analytic models, the analyst usually can complete only a limited number of system performance measures. With simulation models, the data generated can be used to estimate any conceivable performance measure.
- In some instances, simulation is the only means of deriving a solution to a problem.

Disadvantages:

- Simulation models for digital computers may be costly, requiring large expenditures of time in their construction and validation.
- Numerous runs of a simulation model are usually required and this can result in high computer costs.
- Simulation is sometimes used when analytic techniques will suffice. This situation occurs as users become familiar with simulation methodology and forget about their mathematical training.⁵

There are several basic features which are desireable for a dynamic availability analysis model. These are:

1. The model must contain general component models.
2. Component history must be traceable to provide dynamic system information.
3. Systems should be constructable by linking general component models.
4. Interactions between components must be modelled.
5. Scheduling of system changes at specified times must be possible.

⁵ Taken from reference B-4 page 4, the original information comes from reference S-1.

6. For systems containing continuous process variables, a continuous simulation capability is necessary.

Of these six basic characteristics, the first one is easily obtainable by any programming language. To create general component models it is necessary to define input and output parameters to the component, and a rule, or set of rules, which provide a means of determining the component output and state based on input information. Figure 3.1 shows a general component model. It can be thought of as a box into which signals are fed and an output emerges. In addition to signals, information concerning failure and repair rates must be entered. To provide dynamic system information the signals must be able to change value as a function of time. All of these features are easily programmable in any computer language such as FORTRAN or PASCAL.

The second necessary feature is the ability to track component history. Figure 3.2 shows a time line of performance for a specific component. The component is operational for a random length of time and then experiences a failure. There is a random repair delay modeled indicating passage of time before the failure is detected, and then once the failure is found, repair is begun. The component repair time is also random and once the component is repaired it is returned to its operational state. Note that, in general, the component need not be returned to its original state. To track the availability of systems composed of such components it is necessary to have features of the program language which allow for integration over time to determine the average system unavailability, and which allow for sampling the system or component status at selected instants in time to determine the dynamic system unavailability at discrete time points.

To allow the treatment of process variables (which are the fundamental

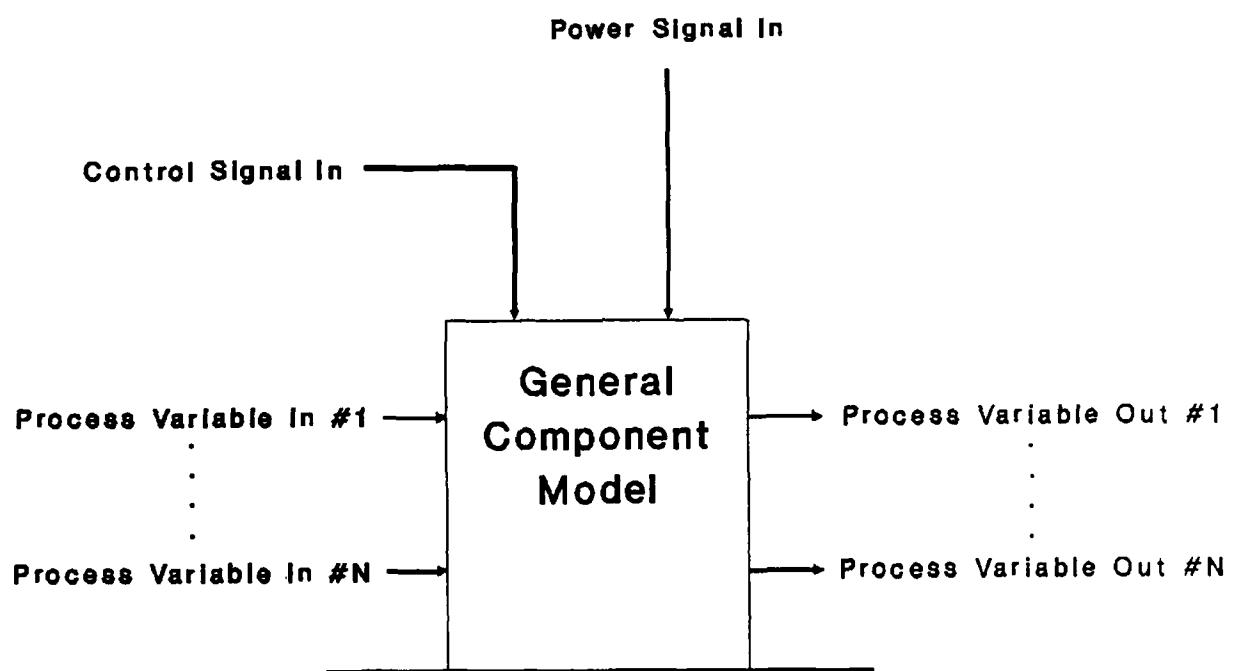


Figure 3.1 General Component Model

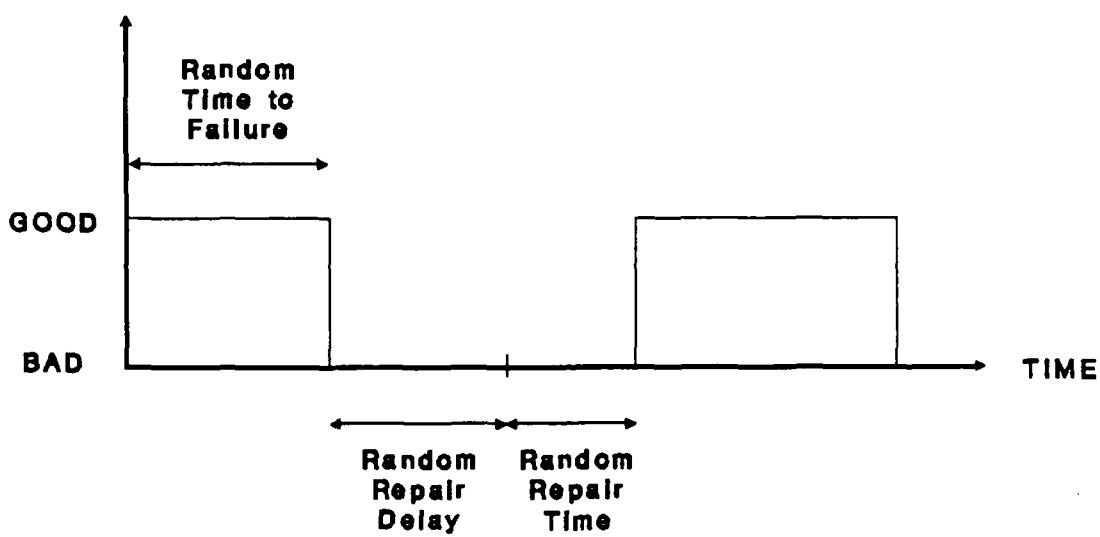


Figure 3.2 Component History

quantities determining system "success" or "failure"), it is necessary to be able to link all of the component models and allow them to interact with each other through the process variables. This requirement means that the output signal from one element will be used as the input signal to another component. This will cause the second component to react to whatever change may have occurred in the first component. In this manner, the failure (or change of state) of one system component can be propagated through the system. By linking the components in this fashion it is possible to create an entire system model out of the general component models. By requiring the components to change state based on their inputs the interaction between components will be modelled. Since in some systems it may be possible to produce loops of elements, it becomes necessary to continue propagating changes through the system in a cyclic fashion until no further changes occur.

To simulate a dynamic system it is necessary to simulate the passage of time. This can be done by two different methods. The first is by discrete event simulation. In this method a queue is created into which events are entered along with their scheduled occurrence times. For example, a command signal causing a valve to close can be scheduled to occur at a specified time, or a pump could be scheduled to be placed in a standby condition to simulate the performance of maintenance. At a separate time, the valve may be given the command to open or the pump could be placed back in an operational state. Numerous such events can be scheduled and entered in the queue; events in the queue are ordered by their occurrence times. In discrete event simulation, the simulation clock is started and time is advanced to the time corresponding to the first event in the queue. This event is executed and the changes propagated through the system. Once the

system has reached a steady state indicating that all consequences of the initial change have been realized, the clock is advanced to the time of the next event in the queue. Operation continues in this manner until there are no more entries in the event queue. This type of event simulation assumes that no changes occur in the system between the scheduled discrete events. This will not be the case for systems involving continuous variables and another simulation approach is necessary for such systems.

In continuous event simulation certain system parameters are continuously variable. This is true for process control problems, or any analyses which involve temperature, pressure, flow rates, or fluid levels as process variables. To simulate this type of system, time is advanced by a small time step and all system parameters are updated. If a component has changed state during the time step, this change is propagated through the system in the same manner as for discrete event simulation. After all changes have been made, the simulation clock is again advanced by the time step and all changes are calculated. Simulation is continued in this fashion until time has been advanced to the end of the desired simulation period.

From the above discussion it is seen that there are three perspectives from which simulation languages can be constructed and these are discussed in reference B-5. These include process-interaction, event-scheduling, and continuous simulation. Process interaction provides modelling of system components as processes (groups of related events such as component failure and component repair) and then relates these processes to each other in a manner which allows the components to interact. Event scheduling allows for events to be scheduled in a queue for occurrence at a specified time during a simulation as is used for a purely discrete event simulation approach. Continuous simulation is used when variables of a continuous nature are to

be considered, in which case discrete event simulation is not appropriate. Any simulation language may use one or a combination of all of these approaches. The most powerful language to use from a systems reliability analysis standpoint would clearly be a language that employs all three positions, since a process interaction capability is desireable to easily model component interactions, event scheduling is desireable to cause the occurrence of such discrete events as system maintenance and testing, and continuous simulation is necessary to treat systems containing continuous variables.

Any of the three approaches discussed above can be programmed in general programming languages such as FORTRAN or Pascal. However, in recent years many program languages have been developed specifically for simulation applications. Some of these languages are based on languages like FORTRAN while others have been developed specifically for simulation problems. One example of discrete event simulation using a standard program language can be found in SIMTOOLS. This product, described in reference S-2, is a collection of data structures and routines which allow the writing of discrete event simulation programs in Pascal using the event view. This method may be simpler for individuals already familiar with the Pascal language, however it may not be as efficient as simulation with the languages which have been designed specifically to treat simulation problems.

Many products are currently available for discrete event simulation. Several of these products are compared and discussed on reference B-4. Their approaches and modeling characteristics are summarized in Table 3.2 which gives a comparison of five languages including FORTRAN to illustrate which programs could be used to perform different simulation functions. It can be seen that as far as modeling approach is concerned, SIMSCRIPT II.5 is

extremely versatile and could be used to model all types of random variables and event occurrences in a Monte Carlo simulation model. SLAM II is a simulation language which also allows for programming of all modeling approaches and could prove to be an equal alternative to the SIMSCRIPT II.5 language for coding of the program developed for this work. The other example languages shown do not have the adaptability to perform all the tasks modeled in the DYMCA M program and the modified DYMCA M program (TANK) described in Chapter 5.

It should be noted that although Table 3.2 includes FORTRAN for comparative purposes, it is certainly possible for anyone so inclined to create there own simulation language equal to, or better than, those mentioned here based on FORTRAN or Pascal. It is possible to write a FORTRAN program oriented to solving any type of system and using any one, or all three, of the modeling approaches, and although random sampling is not built in, there are several scientific subroutines available for random number generation in FORTRAN. In the following section the basic features of the SIMSCRIPT II.5 simulation language will be discussed to describe the features available for the DYMCA M dynamic simulation model.

Table 3.2
Comparison of Languages for Discrete-Event Simulation⁶

| Criteria | Language | | | | |
|---|-----------------|-----------|---------------------|-----------|---------------------|
| | FORTRAN | GASP IV | SIMSCRIPT II,5 | GPSS/H | SLAM II |
| Ease of learning | Good | Good | Good | Excellent | Excellent |
| Ease of conceptualizing a problem | Poor | Fair | Good | Excellent | Excellent |
| Systems oriented toward | None | All | All | Queueing | All |
| <u>Modeling approach</u> | | | | | |
| Event-scheduling | No ⁷ | Yes | Yes | No | Yes |
| Process-interaction | No ⁷ | No | Yes | Yes | Yes |
| Continuous | No ⁷ | Yes | Yes | No | Yes |
| <u>Support</u> | | | | | |
| Random sampling built in | No | Yes | Yes | Yes | Yes |
| Statistics gathering capability | Poor | | Excellent Excellent | Good | Excellent |
| List-processing capability | Poor | Good | Excellent | Good | Good |
| Ease of getting standard report | Poor | | Excellent Fair | | Excellent Excellent |
| Ease of designing special report | Fair | Good | Excellent | Good | Good |
| Debugging aids | Fair | Good | Excellent | Good | Good |
| Computer runtime | Excellent | Good | Good | Good | Good |
| Documentation for learning language and for reference | Very Good | Very Good | Fair | | Very Good Very Good |
| Self-documenting code | Poor | Good | Good | Excellent | Good |
| Cost | Low | Low | High | High | Medium |

⁶ This table is Table 3.8 taken from Reference B-4.

⁷ Modelling structures not built-in, but can be developed.

3.2.2 SIMSCRIPT II.5

There are many references available for explaining the SIMSCRIPT II.5 language and programming techniques for developing simulation models. The text by Law and Larmey (ref. L-4) is a beginning handbook for understanding the language. For a more detailed description on programming procedures, reference R-1 by Russell should be consulted. This latter book explains more of the complicated modeling methods than the introductory text. Other references used in development of the DYMCA model include, reference C-1, reference C-2, and reference F-1. All three of these texts provide useful information for understanding the use of SIMSCRIPT commands and modeling techniques.

SIMSCRIPT II.5 is a general programming language which facilitates the development of a discrete-event simulation model. It allows for both process interaction and event-scheduling points of view, or a combination of the two, in simulation modeling. A language extension in current versions allows for continuous simulations. In addition, it also has powerful scientific computing and list processing capabilities which when used to their fullest degree can be more efficient, for a programmer, than FORTRAN. A unique feature of the SIMSCRIPT program is that it can be written in English-like statements. As a simulation language in comparison to FORTRAN, SIMSCRIPT performs many complicated automatic maintenance tasks and report generation functions. It is very well suited for all types of Monte Carlo simulation problems.

Before proceeding, several SIMSCRIPT terms need defining to provide a basic understanding of simulation programming using SIMSCRIPT. For more complete descriptions the references noted previously should be consulted. The basic terms to be described here include: SCHEDULING, ENTITY, PROCESS,

ATTRIBUTE, and SETS.

SCHEDULING refers to the discrete event feature of SIMSCRIPT. An event queue is created and events are placed in the queue (scheduled) along with their time of occurrence. The events in the queue are arranged in the order of their occurrence time and executed in that order. Time then is advanced to the occurrence time of the next event in the queue. The queue is dynamic; as simulated time progresses, new events may be scheduled and other (previously scheduled) events removed from the queue. For example, a component failure can be scheduled to occur at a certain time. Once the failure has occurred, an event representing repair completion can then be scheduled. As an example of removing events from the queue, an event can be scheduled at the beginning of a simulation which restores all components to as-good-as-new condition at a specified time. This event can remove all scheduled component failures from the queue. Later in the simulation, the failures can be rescheduled to occur at later times.

An ENTITY is a program variable and has a memory location allocated to it once it is created. Entities are of two types, permanent and temporary. Permanent entities are created once, at the beginning of the program, and exist throughout program execution. Temporary entities are created only when needed and memory can be made available again for other variables by destroying the temporary entity once it is no longer needed. This provides a means of keeping data structures contained in computer memory to a minimum, thus providing for more efficient program operation. Several identical entities can be created by using a pointer or indexing variable. For example, if a simulation is to contain 10 valves, the following lines of code could be used to create them:

```
reserve pointer(*) as 10
for i equals 1 to 10
do
    create a valve called pointer(i)
loop
```

Then to refer to a specific valve the pointer can be used.

A PROCESS is a special SIMSCRIPT entity which has memory associated with it in the same manner as a temporary entity. It can have several identical instances created. For example, if a component is modeled as a process, several identical processes can be created, one associated with each component. The most important feature of a process is that it has a subroutine associated with it which can schedule events and interrupt other processes. A process subroutine can also contain statements which cause the execution of the routine to be suspended, and an event notice to be placed in the event queue to cause the process routine to continue execution at a later scheduled time. If a component is modeled as a process, then the failure of the component can be scheduled by the process and process execution suspended until this time has been reached. Once the failure time has been reached, the component process again begins execution in the line of code following the failure scheduling. Here a repair delay can be defined and execution suspended until the scheduled delay time has passed. Then repair can be scheduled in the same manner. A process can also create other processes or temporary entities.

All entities and processes can have ATTRIBUTES associated with them. This is a way of creating a data array. For instance, a pump can be defined as an entity. Several pumps may be created. Associated with each pump there may be a demand failure probability, a failure rate, a repair rate, etc. These characteristics can be defined as attributes of the pump entity and thus when a pump is created, memory storage is also allocated for the array

of characteristics associated with it. Processes can also have attributes in the same manner.

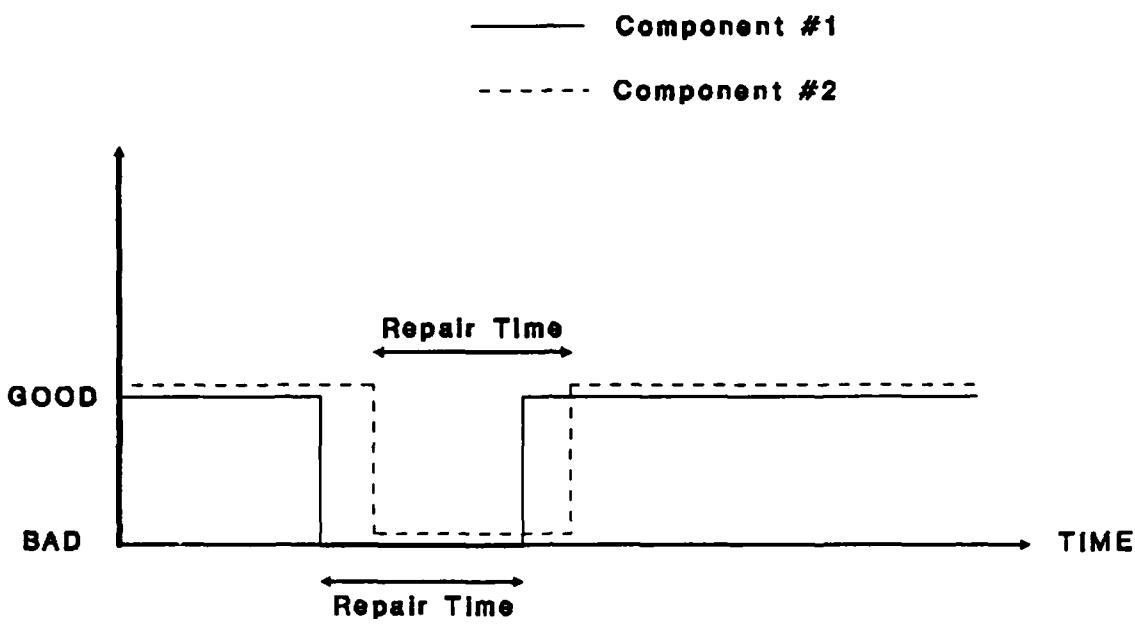
SETS are an important SIMSCRIPT feature. Several items which are of the same type can be grouped as members of a set. These members may be entities or processes, but must be one or the other, in a given set. For example consider a system containing 100 different input and output signals from ten system components. Several of the signals may be input signals to a given component. A signal set can be defined to group these signals. The set will be "owned" by the component process, and the input signals will "belong" to the set. Thus in SIMSCRIPT terminology, all sets must have an owner and may have any number of members which belong to the set.

SIMSCRIPT also has useful statistics features available for evaluating system simulation. The two basic commands are TALLY and ACCUMULATE. The Tally command is used to compute statistics of a distribution, such as the mean and variance, at specified instants of time. The distribution can be an array variable. The Accumulate command tracks the behavior of an entity over the duration of a simulation. It performs integration with respect to time and can be used to determine the time-averaged behavior of a system entity. By properly defining the possible component states, this feature can be used directly to calculate time averaged system unavailability information.

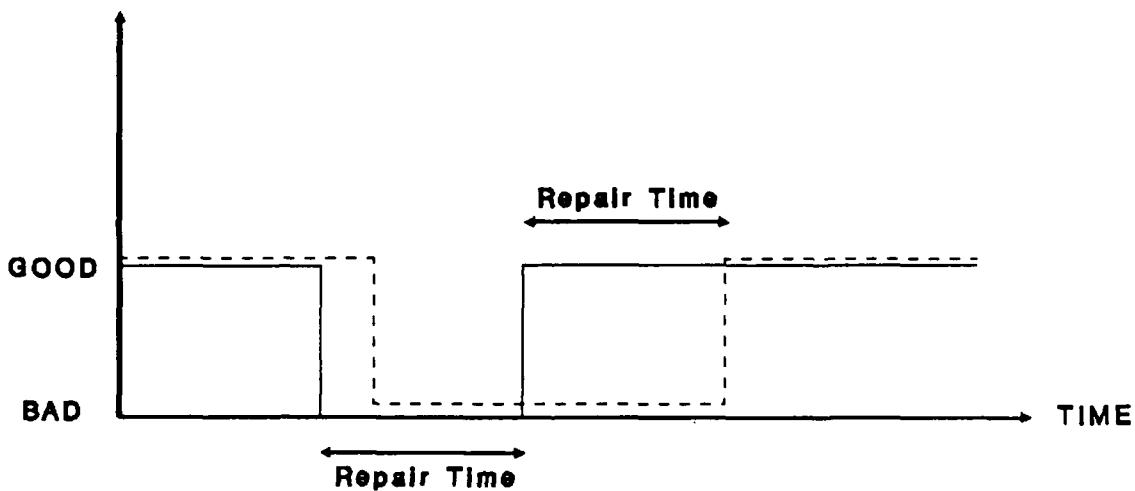
The process-interaction approach to simulation modeling allows SIMSCRIPT to be very useful in the analysis of complicated phased mission problems. Components can be modeled as processes thus allowing each component to control its own time dependent behavior. Failure and repair procedures can be included in the component process subroutine to provide scheduling of failure and repair times. By modeling testing and maintenance

as separate processes it is possible to correctly model random testing and maintenance events interrupting component operation and then restarting the components once they are completed. If it is desireable to limit repair resources, such as by limiting the number of components under repair at any given time, or if random repair delays are to be incorporated based on the number of components presently failed, it is possible to create a "repair supervisor process." This process could be used to schedule repair processes by interrupting and rescheduling selected component events. Event scheduling techniques do not readily allow this flexibility, since scheduling of certain events, such as repair completion times and testing termination intervals, are dependent on the occurrence of other random events and are therefore best modeled by processes rather than sequences of events.

As an example, consider the time line of Figure 3.3A. A system is composed of two components with exponential failure distributions and constant repair time. Using discrete event scheduling, the failure and repair times can be scheduled for both components before the start of the simulation based on their failure rate and repair time data as shown in Figure 3.3A. The occurrence times are independent and do not interact with each other. However, if repair resources are to be limited such that only one component can be under repair at a time, then the event scheduling shown in figure 3.3A is not adequate. A repair supervisor process can be used which counts failures of components and schedules repair events. Then when the first component fails, the repair process will immediately schedule the repair time. When the second component fails, if the first component is still under repair, then the process can wait until repair of the first component is complete before repair is begun on the second component. This is illustrated in Figure 3.3B. This type of component interaction requires



a.) Component Repair Modeling, Two Repairmen



b.) Component Repair Modeling, One Repairman

Figure 3.3 Limited Repair Resource Modeling

some form of process interaction simulation approach to model adequately.

The continuous capability is important for analysis of problems which have continuous random variables. It allows for straightforward analysis of process controls systems and like structures which include such continuous parameters as temperature, pressure, flow rates, or fluid levels. This program capability is taken advantage of in Chapter 5. The report generating functions are also extremely useful as they allow easy calculation of such system parameters as all statistical values and all time averaged quantities of interest. The SIMSCRIPT II.5 simulation language provides many other valuable programming features useful for developing Monte Carlo simulation models and several of these shall be evident in later sections of this work. For more complete information the references should be consulted.

3.3 Program Objectives

The DYMCA (Dynamic Monte Carlo Availability Model) base simulation program was developed with the three primary objectives. These objectives are: 1) provide a model which is able to analyze the time-dependent unavailability of dynamic systems, 2) provide a model which is easy to construct and interpret, and 3) provide a base model which can easily be expanded to incorporate additional features as needed.

To analyze the time-dependent unavailability of a system it is necessary to consider two basic program abilities. The program must incorporate component repair and there must be a program feature which allows the scheduling of external events. Consider the time line of Figure 3.4. During the course of a simulation, two types of events must be scheduled to cause changes in components of the system. These are internal events and external events. Internal events are failure and repair times of individual components. Scheduling of these events is controlled by the individual

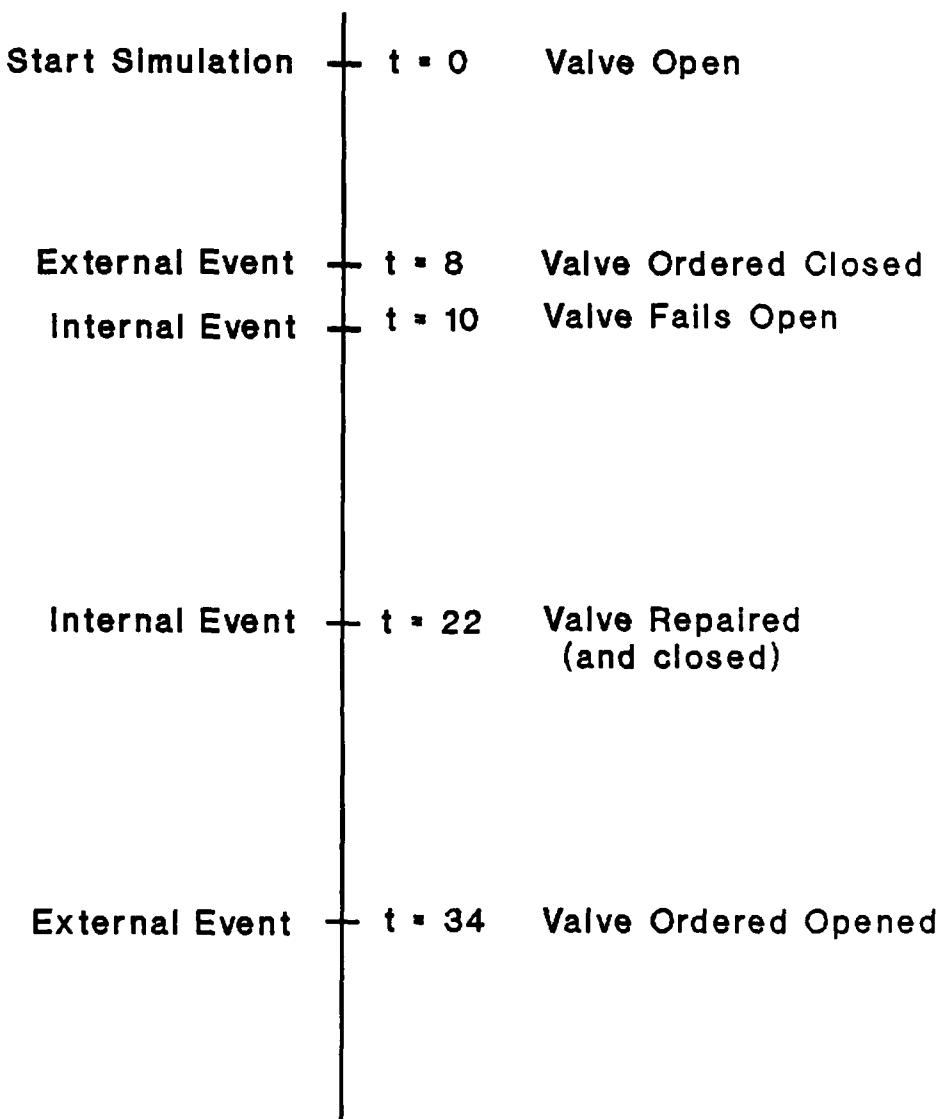


Figure 3.4 Event Scheduling Approaches

component processes. External events are scheduled at the start of a simulation by the user. In the example shown these types of events could include such occurrences as changing the state of a valve from closed to open. The time-dependent availability of a system will depend on both types of events.

Another time-dependent feature desired, is the incorporation of interactions between components based on process variables. When one component in a system changes state, it may have an impact on other components in the system. Thus the change must be propagated through the system to determine what effect there is on other components. For example, an open valve may be supplying water to an operating pump. If the valve fails closed, flow to the pump is stopped. If the pump does not stop on its own, it will fail. Even in the absence of directly caused failures such as this, it is necessary to propagate interactions to obtain the state of all process variables.

To provide a model which is easy to use and interpret, it is necessary to provide component models in the program which correspond directly with physical components. This is a feature provided by both GO and GO-FLOW as was discussed in Chapter 2. It is also the approach adopted by Apostolakis, Salem, and Wu when constructing fault trees based on decision tables (ref. A-4). The problem with GO was that it is designed to be used in static analysis cases. In the case of GO-FLOW, which is designed for time-dependent analysis, the method does not incorporate repair directly into its operators and can therefore not directly treat the failure and repair cycles of components. The equations of the GO-FLOW operators in Table 2.1 demonstrates this fact.

The model to be developed should have a one-to-one correspondence

between physical components and model components and connections between components in the model should reflect actual component interactions in the physical system. Power and control signals to components must be modeled as well as process variables. The set of rules governing the possible changes in component state as a function of the various input signals supplied to them should clearly reflect the physical changes actually experienced.

Since the model to be developed is to be a basic one, it should also be easy to modify so that additional feature may be incorporated. Such features may include non-exponential transitions, dependent repair and failure of components, uncertainty analysis, continuous process variables, and complex interactions. Specific features of the actual model will be discussed in section 3.5.

3.4 Model Assumptions

To implement the general model to treat the behavior of some simple, but realistic systems, a number of assumptions are made as detailed in this section. Many of these assumptions can be relaxed at a future date if more work is to be done on the DYMCA dynamic simulation model. They do not represent restrictions of the simulation language or the program, but merely modeling simplifications. Where applicable comments will be made concerning relaxing the assumptions.

The base model considers exponentially distributed failure and Weibull distributed repair times. Dependent failure and repair are considered only to the extent that the loss of the process variable to an active component causes it to fail if it is in an operating state, and external events can be used to model shocks which fail several components simultaneously. Uncertainty analysis is not included. Continuous variables are included in the program modification described in Chapter 5. Complex interactions are

also considered, to a certain extent in Chapter 5, as operational states of components are dependent on the level of the continuous process variable.

The output generated by the program is also of importance. The desired output of the DYMCAW program is a print out of the time dependent system unavailability and the average system unavailability over the duration of simulated time. The time-dependent unavailability can be printed in several fashions. If desireable, the user can specify an arbitrary set of time points between time zero (start of simulation) and the terminal time (end of simulation) and enter these time values directly in the input file. Or, if it is preferred, the number of time points desired can be specified in the input file and the program will automatically select the specified number of points, uniformly distributed over a linear or logarithmic scale between the starting time and the end time.

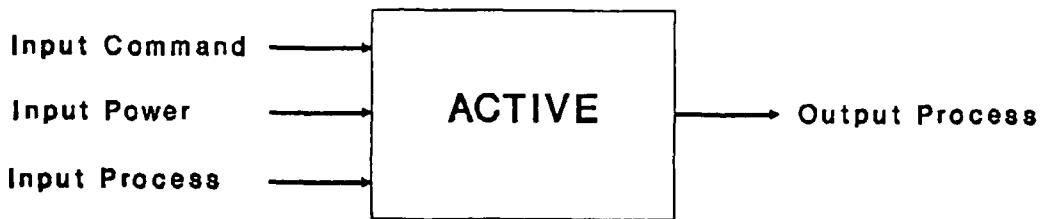
For computing the average system unavailability over the duration of the simulation, the program uses the basic SIMSCRIPT II.5 Accumulate commands. Over each simulated system run the total time the system was unavailable is determined and then divided by the total time. This value is stored for each run and statistics are taken to determine the average and standard deviation of the average unavailability over the number of system runs. The average and standard deviation are then printed in the output file along with selected percentiles of the distribution. If desireable, a simple program modification allows for printing of the average system unavailability for every trial. This is done in one of the example problems discussed in Chapter 4.

To model system components, five component types were chosen. These include valves, check valves, switches, and generic active and passive components. Component types are defined by the number and type of input

signals, by the possible internal states of the component, and by the rules used to process the input/output signals as a function of the component state.

A large number of engineering components can be modeled effectively using these basic elements. Active components, valves, and switches have a minimum of three inputs which include a power signal, a command signal, and at least one process input. Passive components have a minimum of one input. They require at least one process input and do not require power or commands. All components can have any number of process outputs. Figures 3.5 to 3.9 provide diagrams and rule tables describing the five component types. The rule tables are taken directly from the model program listing of Appendix B. Generally, at the start of a run, no component is initially in a failed state. Note that it is a simple matter to use an external event to change a component to a failed state at time zero.

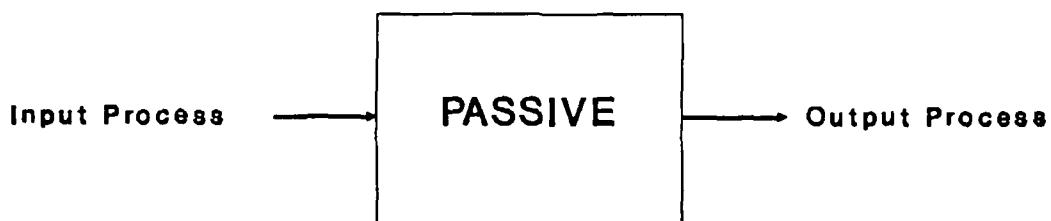
Changes can be forced on the system at any time through the use of external events. These external events can be scheduled to occur during the simulated system operating period and can be used to change the state of components or to change system signals, such as changing a command signal to tell a pump to turn on or off. The current model requires the times of such occurrences to be known before the start of the simulation and included in the input file. The programming language, however, will allow for the random scheduling of these external events. If this is desireable at a later date, it simply involves creating a process routine (similar to the repair supervisor routine) which schedules events in a random fashion. This type of routine may be useful for treating unscheduled testing and maintenance.



Decision Table

| Case | Command Input | Power Input | Process Input | Initial State | Final State | Process Output |
|------|---------------|-------------|---------------|---------------|-------------|----------------|
| 1 | - | - | - | failed | failed | no |
| 2 | - | no | - | standby | standby | no |
| 3 | stop | yes | - | standby | standby | no |
| 4 | none | yes | - | standby | standby | no |
| 5 | start | yes | no | standby | standby* | no |
| 6 | start | yes | yes | standby | standby* | no |
| 7 | - | no | - | operating | operating | yes |
| 8 | stop | yes | no | operating | standby | no |
| 9 | stop | yes | yes | operating | failed | no |
| 10 | stop | yes | yes | operating | standby | no |
| 11 | none | yes | yes | operating | operating | yes |
| 12 | start | yes | no | operating | failed | no |
| 13 | start | yes | yes | operating | operating | yes |
| 14 | - | - | - | standby* | standby* | no |
| 15 | - | no | - | operating* | operating* | no |
| 16 | - | yes | no | operating* | failed | no |
| 17 | - | yes | yes | operating* | operating* | yes |

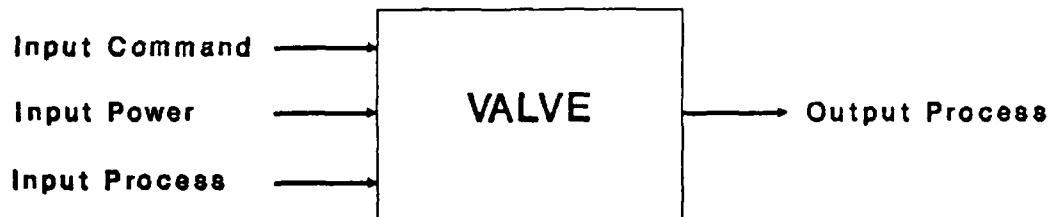
Figure 3.5 Active Component



Decision Table

| Case | Process Input | Initial State | Final State | Process Output |
|------|---------------|---------------|-------------|----------------|
| 1 | - | failed | failed | no |
| 2 | no | standby | standby | no |
| 3 | yes | standby | failed | no |
| 4 | no | operating | operating | yes |
| 5 | yes | operating | standby | no |
| | | | operating | yes |

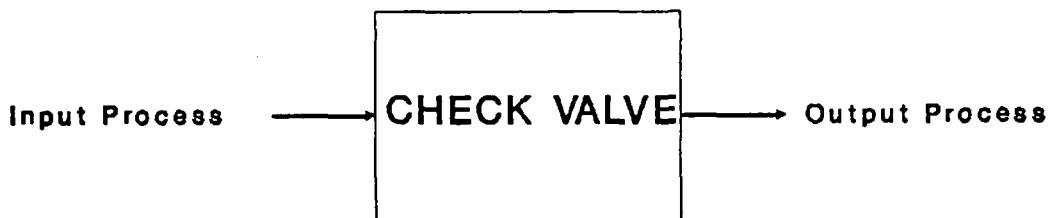
Figure 3.6 Passive Component



Decision Table

| Case | Command Input | Power Input | Process Input | Initial State | Final State | Process Output |
|------|------------------|----------------|------------------|------------------|----------------|-------------------|
| 1 | - | - | - | failed_open | failed_open | no |
| 2 | - | no | - | open | open | no |
| 3 | open | - | - | open | open | no |
| 4 | none | - | - | open | open | no |
| 5 | close | yes | no | open | failed_open | no |
| 6 | close | yes | yes | open | closed | no |
| 7 | - | - | no | failed_closed | failed_closed | no |
| 8 | - | - | yes | failed_closed | failed_closed | yes |
| 9 | - | no | no | closed | closed | no |
| 10 | - | no | yes | closed | closed | yes |
| 11 | open | yes | no | closed | failed_closed | no |
| 12 | open | yes | yes | closed | open | no |
| 13 | none | - | no | closed | closed | no |
| 14 | none | - | yes | closed | closed | yes |
| 15 | close | - | no | closed | closed | no |
| 16 | close | - | yes | closed | closed | yes |

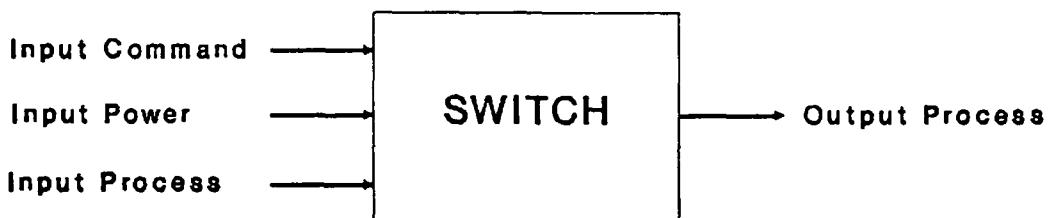
Figure 3.7 Valve



Decision Table

| Case | Process Input | Initial State | Final State | Process Output |
|------|---------------|---------------|-----------------------|----------------|
| 1 | - | failed_closed | failed_closed | no |
| 2 | no | closed | closed | no |
| 3 | yes | closed | failed_closed open | no yes |
| 4 | no | failed_open | failed_open | no |
| 5 | yes | failed_open | failed_open | yes |
| 6 | no | open | failed_open closed | no no |
| 7 | yes | open | open | yes |

Figure 3.8 Check Valve



Decision Table

| Case | Command Input | Power Input | Process Input | Initial State | Final State | Process Output |
|------|---------------|-------------|---------------|---------------|---------------|----------------|
| 1 | - | - | - | failed_closed | failed_closed | no |
| 2 | - | no | - | closed | closed | no |
| 3 | close | - | - | closed | closed | no |
| 4 | none | - | - | closed | closed | no |
| 5 | open | yes | no | closed | failed_closed | no |
| 6 | open | yes | yes | closed | failed_closed | no |
| 7 | - | - | no | failed_open | failed_open | no |
| 8 | - | - | yes | failed_open | failed_open | yes |
| 9 | - | no | no | open | open | no |
| 10 | - | no | yes | open | open | yes |
| 11 | close | yes | no | open | failed_open | no |
| 12 | close | yes | yes | open | failed_open | yes |
| 13 | none | - | no | open | closed | no |
| 14 | none | - | yes | open | open | no |
| 15 | open | - | no | open | open | yes |
| 16 | open | - | yes | open | open | no |

Figure 3.9 Switch

Component failure times are considered to be exponentially distributed. Component repair rates are assumed to be Weibull distributed. The SIMSCRIPT II.5 language allows for many types of distributions, therefore it is an easy matter to change distribution types if others are more appropriate for certain applications. These changes can accommodate such time-dependent effects as component aging.

Also concerning component transfer rates, the possibility of demand failures of active components, valves, and switches are considered. These data are entered in the input file⁸ and applied to cases of the indicated component failing to transfer in either direction. For instance, a valve can fail to open when it receives a signal to open or it can fail to close once it receives a signal to close. This may be a problem if the two failure mode probabilities are of different magnitude, but this can be easily rectified by making minor changes to the program and the input file.

Currently there is no capability to consider repair delays but as is discussed in an example of Chapter 4 these are easily including in the REPAIR.SUPERVISOR routine. If it is necessary to always consider repair delays for components, then it may be desireable to modify the component routine of the program to include a repair delay and to change the input file so that a repair delay time, or the parameters for a repair delay distribution can be entered.

Concerning process signals in the program which will represent such system characteristics as fluid flow, pressure, temperature, or electric current, there is currently no provision in the model to determine signal magnitudes. It is assumed that the existence or non-existence of the signal

⁸ The DYMCA program input file is described in Appendix A.

is enough to establish the state of components or of the system. In fact all components can have any number of process inputs and process outputs and where inputs are concerned, if the component has at least one input signal indicating it is on, then if the state of the component is correct, all output process signals will be "on". For the case of a two-out-of-three system (an example is considered in Chapter 4) it is possible to modify the program by changing the input requirements to a valve so that it does not produce output unless it has at least two input signals. This, however, is not a satisfactory solution, in general, if process signal strength is important in the system analysis. Again it would require modifications to all component routines and the input file to accommodate the notion of signal strength, but this could be accommodated by the SIMSCRIPT language and this is a minor point. Currently, there is no limit to the number of input or output process signals from any given component, so clearly, if signal strength is to be considered, an algorithm must also be included which divides the input signal strength between the available outputs based on the physics of the system.

There are many basic assumptions in the DYMCA M dynamic simulation model, but most of them can be relaxed if time and effort is taken to modify the DYMCA M program. The SIMSCRIPT language provides numerous capabilities and can accommodate almost any feature that may be desireable in a Monte Carlo simulation model. It is readily apparent that Monte Carlo dynamic simulation modeling can be a powerful tool for reliability analysis. In the next section the DYMCA M program will be discussed in detail.

3.5 Program Description

In SIMSCRIPT II.5 there are many language features which may not be familiar to those who are accustomed to other program languages. First of

all, every program is composed of many subroutines. Two subroutines which are common to all programs are the "PREAMBLE" and the "MAIN" subroutines.

The PREAMBLE is essentially where all initialization is made for the execution of the program. The MAIN routine is where overall program execution is controlled from. For simple programs, this may be the only routine used other than the PREAMBLE. It is used to call the subroutines and to start and stop the simulation program.

The DYMCA M program contains many subroutines, including the PREAMBLE and MAIN routines. Table 3.3 gives a list of all these routines and their basic purposes. A complete listing of the DYMCA M computer program is contained in Appendix B. The remainder of this section gives a brief description of the purpose for, and operation of, each DYMCA M program subroutine in the context of the program operational flow chart as shown in Figure 3.10.

Several subroutines are executed before the beginning of actual system simulation. The first of these is the Input subroutine. The INPUT routine is used to read the data from the input file and record the information in the appropriate memory locations. In particular, it defines the characteristics of the components to be modeled. This routine is called once during the execution of the program from the MAIN routine. The next routine called from MAIN is RUN.INITIALIZE. This routine uses the input information just read in to link the system components together by filing signals in appropriate input and output sets of various components. It also records appropriate signals and components in files associated with each external event for reference when the external event is executed. This routine also initializes all entities. Variables which are not assigned values are automatically set equal to zero by SIMSCRIPT.

The routine TRIAL.INITIALIZE is called from the MAIN program inside the loop which is executed once for each Monte Carlo trial which is to be run. Its purpose is to reset the state of all components and signals to the initial value they should have at the beginning of execution of the next simulation trial.

Table 3.3**DYMCAM Subroutines**

| <u>Subroutine</u> | <u>Description</u> |
|--------------------------|--|
| PREAMBLE | Defines all Entities and Processes |
| MAIN | Controls overall execution |
| ACTIVE | Controls Active components |
| AVAILABILITY | Process that takes time-dependent data for unavailability |
| CALL.UPDATE | Process that causes delay then calls Update routine |
| CHECK.VALVE | Controls Check Valves |
| COMPONENT | Process to control failure and repair of Components |
| DEMAND.TEST | Determines failure on demand |
| EXTERNAL.EVENT | Process to execute External Events |
| FAILURE.TRANSLATION | Function to determine failed state |
| INPUT | Reads input file |
| PASSIVE | Controls Passive components |
| REPAIR.SUPERVISOR | Process to allocate Repair resources |
| RUN.INITIALIZE | Initializes Variables for Run |
| RUN.OUTPUT | Prints output results to a file |
| SCHEDULE.AVAIL.SAMPLES | Process to cause recording of time dependent unavailability data |
| SCHEDULE.EXTERNAL.EVENTS | Process to schedule External Events |
| STOP.SCENARIO | Stops execution of all processes |
| SWITCH | Controls Switches |
| SYSTEM.UPDATE | Propagates Component changes through the system |
| TRIAL.INITIALIZE | Initializes Variables for a Trial |
| VALVE | Controls Valves |

The next two routines called from inside the loop of the MAIN routine are the scheduling modules. The SCHEDULE.AVAIL.SAMPLES process is used to schedule interrupts in the execution of a simulation run to sample the system unavailability. The sample times specified by the user are entered in the

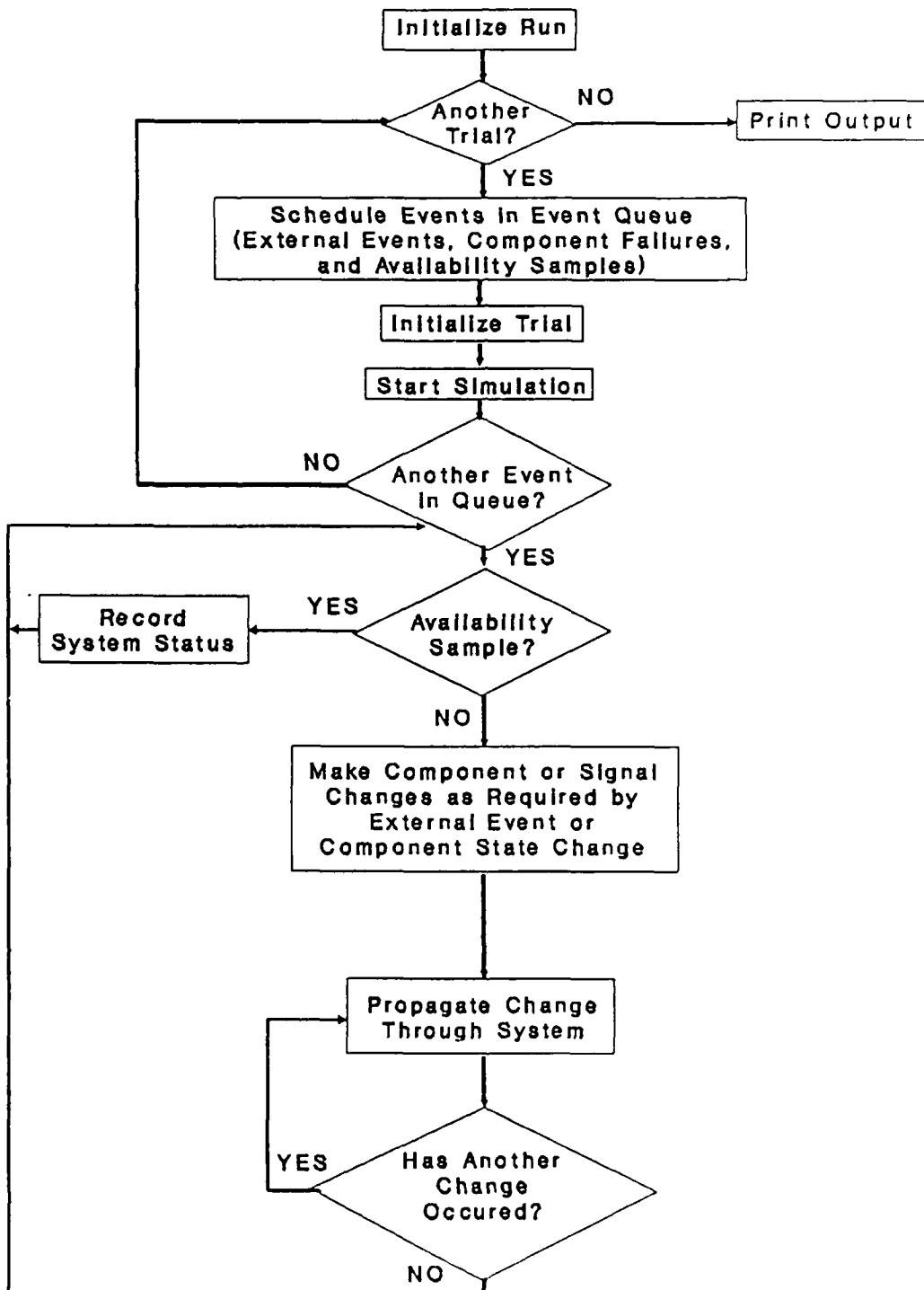


Figure 3.10 DYMCAm Program Flow Chart

event queue for subsequent simulation interruption for system sampling. The actual recording of the availability information is done by the AVAILABILITY process. There is an AVAILABILITY process for each time point specified by the input file and each one collects data for its assigned time point. These samples are totaled for each particular sampling time and divided by the total number of trials in the output routine to determine the system time-dependent unavailability. The SCHEDULE.AVAIL.SAMPLES routine is only to schedule the interruptions for sampling by placing event notices in the event queue.

The SCHEDULE.EXTERNAL.EVENTS process is used to schedule the interrupts in the execution of the simulation run for the processing of external events. It schedules these interrupts to occur at the specified times indicated by the input file. For every external event there is an EXTERNAL.EVENT process. Each EXTERNAL.EVENT process has a component set and a signal set associated with it which specify which components and signals are to change. These changes are performed when the external event is executed and then control is passed to the SYSTEM.UPDATE routine. EXTERNAL.EVENT processes are created by the RUN.INITIALIZE routine along with their associated component and signal files.

Also inside the loop in Main is the STOP.SCENARIO routine. It is used to stop the execution of all processes which have not concluded at the end of a trial and to reset the execution of each component to its original operating condition.

The CALL.UPDATE process exists inside the loop of the main routine to escape a complication associated with the simulation language. Any series of commands executed sequentially without undergoing the simulated passage of time must not contain commands which start and stop the same process or

create and destroy the same entity. It is also not possible to activate the same process twice. The program is designed so that on the initial trial of a run, all component processes are activated at time zero by the RUN.INITIALIZE routine. Thus a notice is put in the scheduled events list which will be executed once the timing routine is begun. One of the first statements in the COMPONENT process is a command to suspend operation, since some components, e.g. standby components, may not be operating at the start of the simulation. Standby components are not allowed to undergo failure in this model and therefore should not have failure times placed in the event queue until they are placed in an operational mode. The components that should be operating are then restarted by the SYSTEM.UPDATE routine.

The problem is that the SYSTEM.UPDATE routine should be executed from the loop of the MAIN routine before the passage of simulated time is begun. This would cause an error since the sequential execution of commands would make it appear that a COMPONENT process has been scheduled to start twice. Therefore the CALL.UPDATE routine is included in the MAIN program loop. Its sole purpose is to wait a short period of time so that the simulation clock is started and all components are in the suspended state before the SYSTEM.UPDATE routine is executed and the operation of selected components is started again.

The SYSTEM.UPDATE routine will be called many time during the execution of a simulation program run and it performs many functions. The first time it is called, it is used only to activate the components which should be operational at the beginning of a simulation. These components will advance from their original suspended states and begin their failure and repair cycles. Thus at the beginning of the simulation each operating component, if it has a non-zero failure rate, it will have a failure time scheduled for

it in the event queue.

At this point the simulation is started. Currently there are three types of events scheduled in the event queue. These are component failures, availability samples, and external events. The simulation clock will be advanced to the time corresponding to the first event in the queue, the notice scheduling the event will be removed from the overall schedule, and the event will be processed.

If the event is an external event then an EXTERNAL.EVENT process will be executed. Components in the external event component set and signals in the external event signal set for this external event will be changed to their new values. Then the SYSTEM.UPDATE routine will be called.

If the event is an AVAILABILITY sample, then the system indicator variable, which indicates whether or not the system is in a satisfactory state, will be tested. The result will be summed with previous and future results for that particular time point, and stored for use in generating the output file. No change to the system is made by this interruption, therefore time is advanced to the next event in the event queue without any changes to the system being performed.

If the event is a component failure, then the COMPONENT process for that particular component will again begin operation. The function FAILURE.TRANSLATION will be called and used to determine the state of the failed component. The failed state will be dependent on the type of component and the initial state, e.g. an open valve will fail closed and a closed switch will fail open. FAILURE.TRANSLATION is an example of the use of the SIMSCRIPT function command which simplifies programming when a series of commands is reused often. The commands in the FAILURE.TRANSLATION function could be placed in the COMPONENT routine without complicating

execution of the program. Once the type of failure is determined, a REPAIR.SUPERVISOR process will be activated and the SYSTEM.UPDATE routine will be called.

The SYSTEM.UPDATE routine is used to connect all system components. It is called any time a component changes state or an external event is activated. It looks for changed signals or components and if it finds a change, it calls the response function (SWITCH, VALVE, etc.) for that particular component or the component which contains the altered signal in its input signal file. If this component changes state, or its output signal changes strength, then it will be necessary to propagate this change through the system. The routine continues to call affected components until no further changes occur. This routine also monitors the overall system state and changes it as necessary to reflect whether the system is available or unavailable as a unit according to the definition provided in the input file.

The SYSTEM.UPDATE routine handles the loops which must occur in a process interaction system. The routine stores the value of all system signals and then looks for changes to this set. If a signal changes value then this is an indication that changes are still occurring in the system. The routine looks for components which have changed state or whose input signals have changed strength and calls the associated response function to ensure the component is in the proper operational state. If it is not, it may change according to its response function and new output signal strengths may be generated. These outputs are inputs to other components, so these components must also be updated. Since the possibility exists for loops to occur in system component structure, once all components have been checked once, the new signals are compared with the old signal strengths. If a difference is indicated, then it is possible that a component is not in its

desired state, thus the affected components are evaluated again. This process continues until the value of all signal strengths at the end of an iteration, equal the value of the signal strengths at the beginning of the iteration, indicating that no component has changed state during the last iteration. Since infinite loops may be possible, a maximum number of iterations is specified, which, if exceeded, causes an error message to be printed.

Another important function of the SYSTEM.UPDATE routine is to reset the "failure clock" for components which change state. For example, whenever an Active component is placed in standby from an operating condition, the COMPONENT process associated with the Active component is reset so that when it begins operation again it will start a new failure clock. This program feature is very important for the analysis of phased mission problems where it is feasible that a single component may be turned on and off several times during a simulation run.

The five routines entitled ACTIVE, PASSIVE, CHECK VALVE, VALVE, and SWITCH are the response functions called by the SYSTEM.UPDATE routine used to determine the state of all system components and the value of their output signals. These routines are used to change the state of components when a new command is received or the strength of an input signal changes. Each routine tests the state of the component and the value of all input signals and compares the results to a set of control "rules" to determine the new component state and the value of all of the component output signals. If the component is Active, a Valve, or a Switch and it has been called upon to change state, then the DEMAND.TEST routine is called to determine if the component has failed or not. The DEMAND.TEST routine's sole function is determine if a demand failure occurs based on the demand failure probability

for the component. Once the tests are performed and the component state is modified, execution is returned to the SYSTEM.UPDATE routine.

After a component has undergone failure and the effect propagated through the system, the REPAIR.SUPERVISOR routine is called. The REPAIR.SUPERVISOR process is not developed at this point to meet all of its potential. This process is currently used to start a repair process once a component is failed. Thus it simply reactivates the component process which controls the repair time calculation for the component. The repair process is activated from the COMPONENT routine whenever a component fails. The listing of the REPAIR.SUPERVISOR process in Appendix B contains a version which immediately starts a repair once a failure has occurred. Line 31, which causes a Weibull distributed repair delay, is not being used (it is "commented" out). It is used in one of the examples of Chapter 4. By changing the values of a and b in : s 23 and 24 it is possible to change the repair delay distribution. However, if different repair delay distributions are desired for different components, then the input file structure and other program characteristics must be changed.

The REPAIR.SUPERVISOR process could also be used to limit the amount of repair resources available. It is a simple matter to count the number of components failed and the number of components under repair by checking the status variable associated with each component. Then if too many components are failed it is feasible to delay repair of some components until repair is finished on other components. It is possible to prioritize repair based on which component has been failed the longest since when a component fails its failure time is recorded. If other prioritization schemes are desired they can be programmed in to the REPAIR.SUPERVISOR process.

The COMPONENT process is used to control the transfer between good and

failed states for all components of the system. There is a COMPONENT process for each system component and these Components are created by the Run.Initialize routine. Within the COMPONENT process there is a section which controls the transfer from operational to failed and a separate section which controls the transfer from failed to operational. Whenever a component changes state the SYSTEM.UPDATE routine is automatically called to propagate the component change through the system as discussed above. Under the current program structure, when a component changes state from operational to failed, the component goes to a suspended state. The repair process is not begun until the REPAIR.SUPERVISOR process reactivates the component.

Once the STOP.SCENARIO event is reached in the event queue, the STOP.SCENARIO process is executed. This process removes all remaining events from the event queue and resets all component processes so that all system processes are ready to begin operation for the next trial. With no events now remaining in the event queue, operation of the program is returned to the MAIN routine which causes the RUN.OUTPUT routine to be called. The Run.Output routine is used to write the program results to an output file. The results provided are of two types. There is a print out of the time dependent unavailability data and there is a list of the average system unavailability distribution. Examples of output files are included in Appendix E and will be discussed in Chapters 4 and 5.

3.6 Chapter Summary

In this chapter the DYMCAm dynamic simulation model has been discussed. The discussion began with a review of some currently available simulation languages. These various approaches of event scheduling, process interaction, and continuous simulation were discussed and it was pointed out that all approaches are valuable to Monte Carlo simulation for complex

systems. It is recognized that SIMSCRIPT II.5 contains programming characteristics allowing for use of all three modeling approaches. Table 3.1 shows a comparison of four simulation languages along with FORTRAN and it is clearly evident that SIMSCRIPT is at least as good as, if not better, than the other languages for Monte Carlo simulation applications to evaluate system reliability. A discussion of SIMSCRIPT II.5 programing features is also included.

Sections 3.3 of the chapter dealt with the program objectives for the DYMCA model and section 3.4 discussed the basic assumptions made. In the final section of the chapter the DYMCA program is described. Its many subroutines are listed and brief explanations given of what the purpose of each routine and process is, to give a general understanding of what the program does. For a complete discussion of input file preparation for the DYMCA program, Appendix A should be consulted. A complete program listing is provided in Appendix B. Specific program features and limitations will also be covered in the discussions of particular problems tested in the examples of Chapter 4 and Chapter 5. The next chapter of this work deals with tests of the basic DYMCA dynamic simulation model.

Chapter 4

Test Runs and Results

4.1 Introduction

The DYMCA program was developed as a basic benchmarking model to demonstrate the capabilities of a simulation approach to solving system availability analysis problems. It does not give results that are necessarily any better than other methods, but it has the advantage that it can solve more complex problems. The capabilities of the basic DYMCA program are:

- 1) It can model external events necessary for phased mission problems.
- 2) It treats exponential failure and Weibull repair distributions.
- 3) It provides dynamic unavailability information about the system and also average unavailability information.
- 4) The base model can be easily modified to treat more complex analysis problems.

In this chapter several basic tests of the DYMCA program are conducted to demonstrate these capabilities. The results obtained are compared with analytical results where applicable, and with numerically generated results in the more complicated examples. A fourth order Runge-Kutta method, obtained from reference P-4, is used to solve the Markov equations for the systems which can be modeled by this approach. The GO-FLOW example is compared with exact results as computed using the GO-FLOW method.

The first problem considered involves a single component with exponential repair and failure times. For this example, which can easily be modeled as a two state Markov system, the governing equations can be solved analytically for comparison. The second illustration also involves a single

component with exponential repair and failure; in addition, it includes a second repair state which also has an exponential transition time. This three state problem can be solved analytically using a Markovian approach.

The third problem involves three pumps in parallel, in series with a valve. Success of the system requires two of the three pumps to operate and the valve to be open. This problem involves a slight change to the program to consider the requirement for flow to be present from two pumps. For this problem there are sixteen possible system states with four of the states leading to system success. To solve the sixteen Markov equations would be difficult to do analytically, therefore the numerical Runge-Kutta integration method is used.

The final example pertains to a GO-FLOW model. This approach is used to solve phased mission problems, therefore a comparison would show the usefulness of the DYMCA M program for solving a phased mission problem. Results are compared with the analytic results obtained from the GO-FLOW solution to the problem. In addition, this problem is used for a sensitivity analysis of the program to demonstrate the variation of the accuracy of the DYMCA M program results with the number of Monte Carlo trials performed.

The chapter concludes with a summary of the performance of the basic DYMCA M dynamic simulation model over the test cases considered. General comments are made concerning the demonstrated capabilities and the accuracy of results. Consideration is made of how this approach compares with other system reliability analysis methods.

4.2 Single Component, Single Repair State

The first example problem to be tested on the DYMCA M program is a very basic example for which an analytical result is readily available. The analytical equation governing the unavailability of the component is:

$$Q(t) = \left[\frac{\lambda}{(\lambda + \mu)} \right] \{1 - \exp[-(\lambda+\mu)t]\} \quad (4.1)$$

where λ = failure rate

μ = repair rate

and

$Q(t)$ = probability component is failed at time t

The illustration is a single component with exponential repair and failure rates. For ease of examination, equal repair and failure rates were considered. The asymptotic value of system unavailability is clearly 0.5 since the component will spend equal time in each of the two possible states.

The DYMCA program computes both instantaneous unavailability of a system to provide the dynamic output, and it computes the average unavailability. Instantaneous availability is computed by stopping the simulation (during each Monte Carlo trial) at a user-specified time and checking the system to see if it is in a failed state. A success state is indicated if the system indicator variable is equal to one, and failure is indicated by a zero. Thus the system indicator value is summed over all of the Monte Carlo trials at each selected time point. A different variable is kept to sum the value for each time point. After all runs are completed, the totals of these variables are divided by the number of trials. If the result equals one, then the system was always available at that time point. If the result is zero, then the system was always unavailable. Numbers between zero and one reflect the probability that at the instant of the time point, the system was available. Thus by subtracting all values from one, an estimate of the instantaneous unavailability is made for each selected time point. The simulation result is an estimator for the exact result given by equation 4.1. The simulation estimates apply at the discrete points

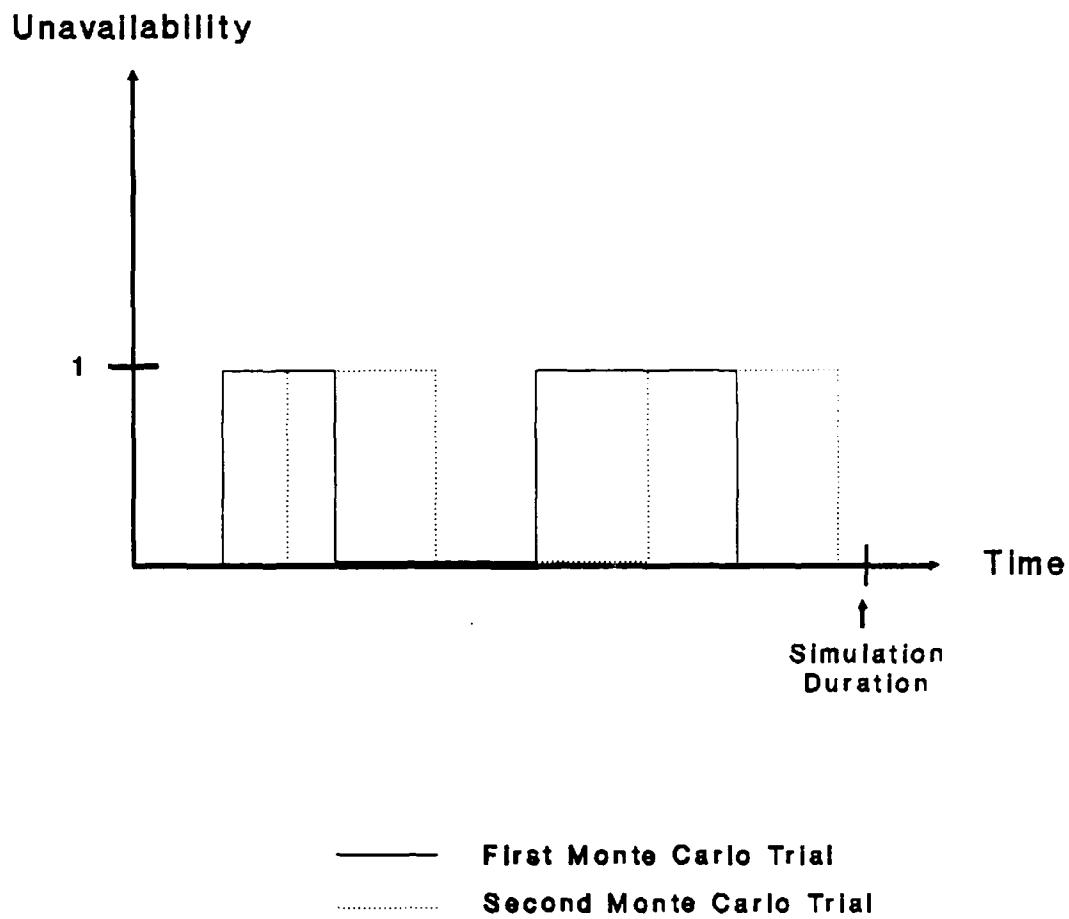


Figure 4.1 Simulation Unavailability Time Line

chosen for analysis in the program.

Average unavailability is calculated over the duration of a simulation. Consider the time line of Figure 4.1. For each trial there will be a distribution similar to the ones shown, but with failure and repair events occurring at different times. For each Monte Carlo trial the area under the curve is integrated and divided by the total simulation duration time. This is an automatic function available in SIMSCRIPT. Since the height of the line in Figure 4.1 is one, the integral of the area under the curve simply equals the total time during the simulation duration for which the system was unavailable. By dividing this result by the total simulation time, an estimate of the average unavailability is obtained. For each trial this result will be slightly different, thus the average unavailability is stored for each trial, and after all trials are completed, a mean, a variance, and selected percentiles of the distribution can be printed. The resulting distribution is an estimator of the exact result given by the equation:

$$Q_{\text{average}} \text{ (for } 0 \leq t \leq T) = \left[\frac{1}{T} \right] \int_0^T Q(t) dt \quad (4.2)$$

To perform the test for proper asymptotic results, the failure and repair rates were chosen to be 0.01 per hour. Thus after approximately 200 hours the system will have reached its asymptotic condition. Each simulation run covers 10,000 hours. For the simple system only 100 Monte Carlo trials were run to give satisfactory results. To show the fluctuations in unavailability about the asymptotic value, the system instantaneous unavailability was printed at every 500 hours of the simulation. To see the average system unavailability the time averaged system unavailability for each trial was printed.

Table 4.1 shows the fluctuation of the instantaneous system

unavailability about the desired value of 0.5. Over the relatively small number of Monte Carlo trials performed we see that there is a rather large fluctuation. This can readily be reduced by increasing the number of trials since the standard deviation of the result is related to the number of trials by one over the square root of the number of trials.

Table 4.1

Single Component, Single Repair State, Instantaneous Unavailability

| TIME | UNAVAILABILITY |
|---------|----------------|
| 0.0 | 0.0 |
| 500.0 | 0.52 |
| 1000.0 | 0.38 |
| 1500.0 | 0.51 |
| 2000.0 | 0.60 |
| 2500.0 | 0.48 |
| 3000.0 | 0.48 |
| 3500.0 | 0.46 |
| 4000.0 | 0.51 |
| 4500.0 | 0.47 |
| 5000.0 | 0.45 |
| 5500.0 | 0.56 |
| 6000.0 | 0.41 |
| 6500.0 | 0.54 |
| 7000.0 | 0.48 |
| 7500.0 | 0.45 |
| 8000.0 | 0.50 |
| 8500.0 | 0.51 |
| 9000.0 | 0.57 |
| 9500.0 | 0.55 |
| 10000.0 | 0.49 |

Using the values of time averaged unavailability for each of the 100 Monte Carlo trials, a graph was constructed showing the distribution of unavailability values. This is shown in Figure 4.2. This figure portrays basically the same information about the model as Table 4.1. The difference is that Table 4.1 provides data that was computed using the instantaneous unavailability estimation procedure discussed in conjunction with equation 4.1 and Figure 4.2 shows the distribution of the time averaged unavailability estimator as discussed in conjunction with equation 4.2. The exact average

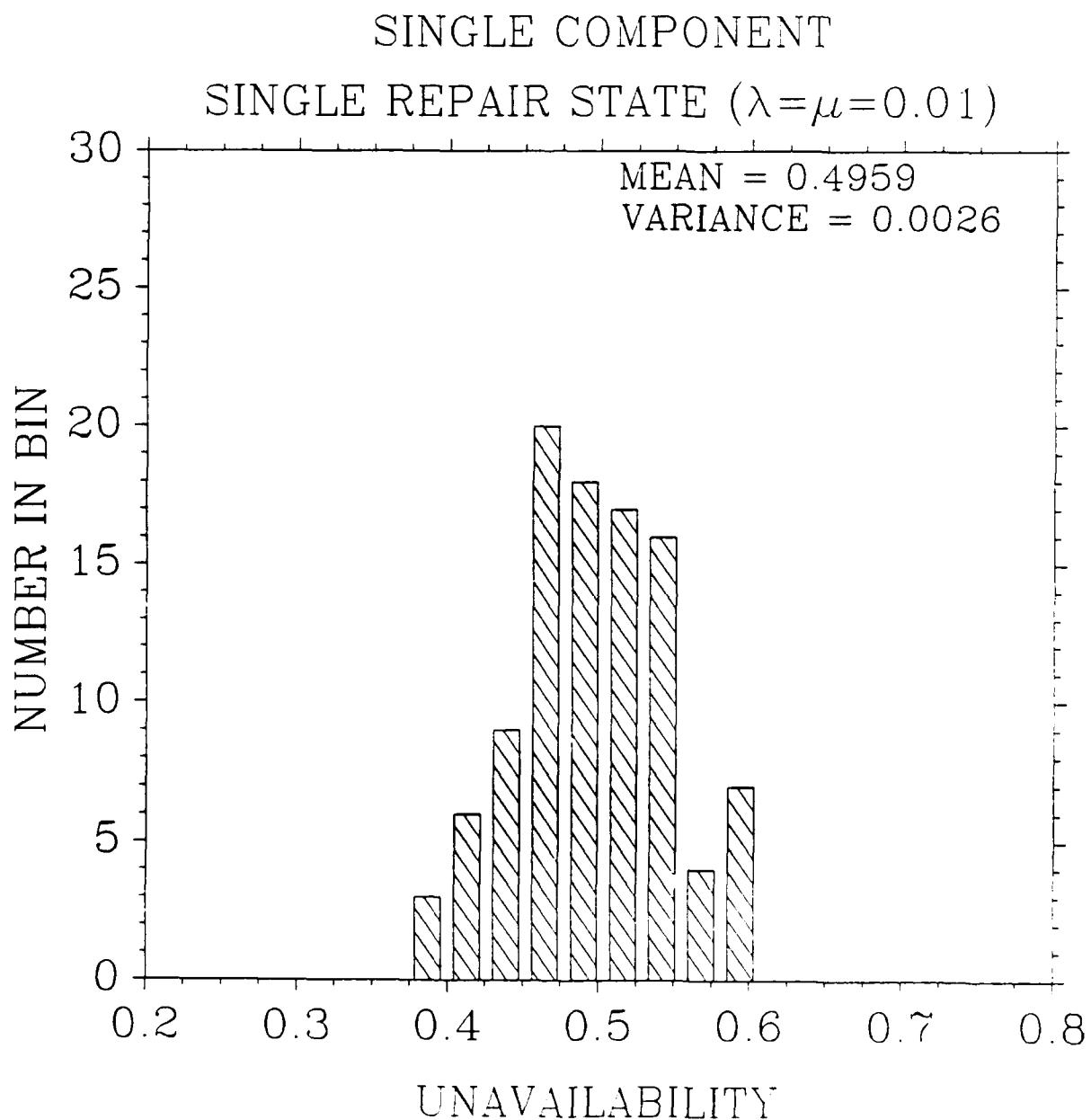


Figure 4.2 Single Component, Single Repair State

Average Unavailability

unavailability can be found by performing the integral in equation 4.2. Doing this integration, where $\lambda = \mu = 0.01$, the result is found to be 0.4975 following 10,000 hours of operation. This result agrees within less than one percent with the mean value of the distribution shown in Figure 4.2. The variance of the distribution indicates a standard deviation of the result of ± 0.05 . For many applications this deviation may be insignificant. The standard deviation can be reduced by increasing the number of Monte Carlo trials performed as shall be demonstrated in a later example.

Another area of interest is whether or not the DYMCA M program provides adequate time dependent unavailability information. Another test was run with the same example problem only over a simulated time period of 200 hours. To reduce the wide variance experienced in the above example the number of Monte Carlo trials was increased to 1000. For comparison the results are plotted in figure 4.3 with the analytic results obtained by Markov analysis of the two state component as shown in equation 4.1.

Figure 4.3 shows that the simulation model provides good time dependent results for this example. At large values of time, however, it is seen that the simulation starts to deviate from the desired results. In fact, for times greater than 200 hours, the simulation continues to fluctuate above and below the desired 0.5 value for unavailability. This, again, is a demonstration of the fact that there will be statistical fluctuations in a Monte Carlo simulation. The fluctuations are smaller the larger the number of trials used.

A final point of concern with a simulation approach to systems reliability analysis is the computer time required to perform the analysis. For this simple one component system the time required to obtain the above results was approximately 30 minutes on an IBM compatible XT machine running

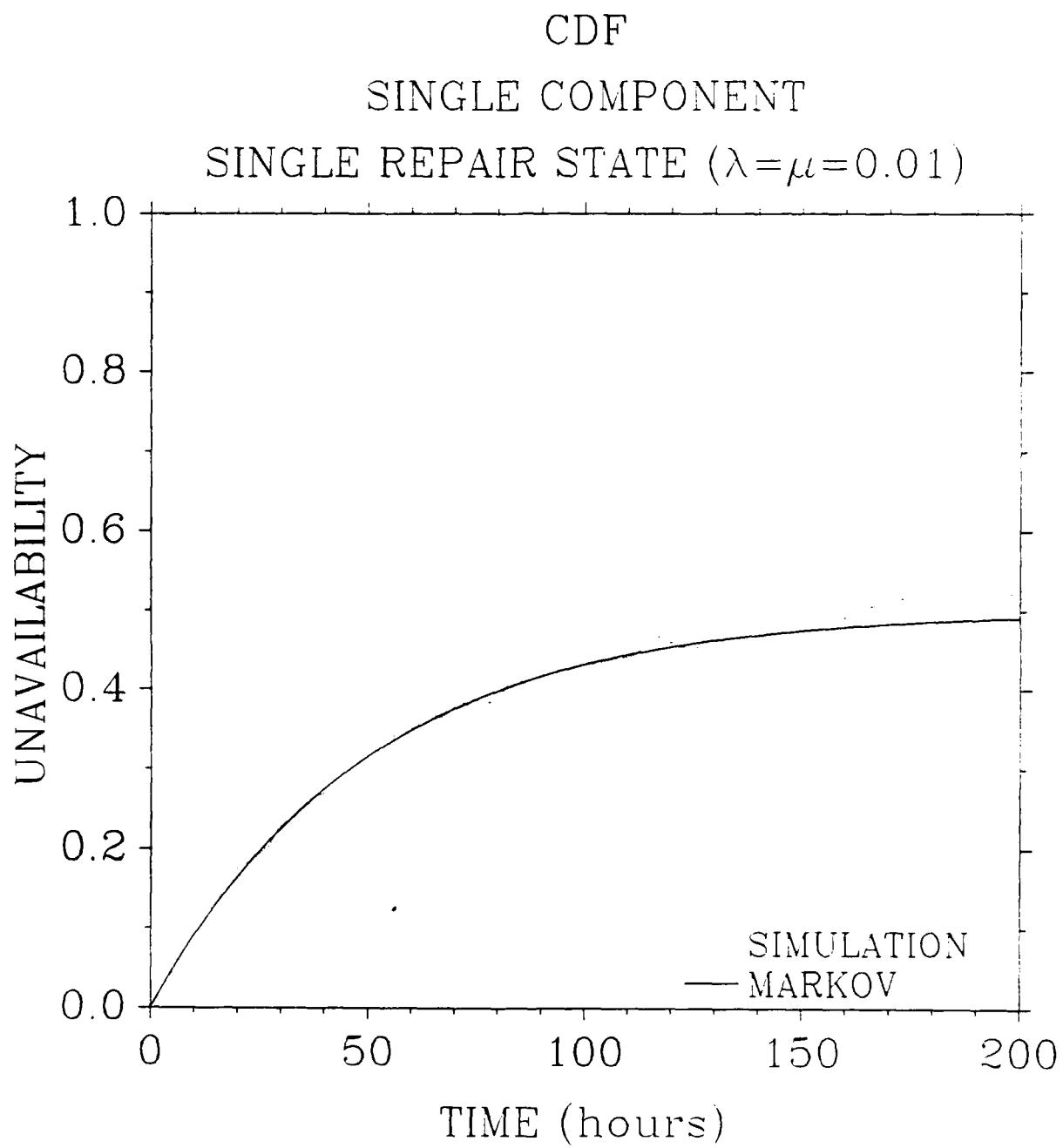


Figure 4.3 Single Component, Single Repair State
Time Dependent Unavailability

at 7.16 MHz. The average unavailability test required a large amount of time due to the long simulated time period of 10,000 hours, which allowed for an average of fifty failure and repair cycles per Monte Carlo trial. The value of fifty is assumed since if the mean failure and repair times are both equal to 100 hours, then every 200 hours the component will, on average, go through a complete cycle of failure and repair. The time dependent analysis required 30 minutes to run even though it simulated a shorter time period, because the unavailability of the system was sampled once every simulated hour (200 points) which slowed down program execution. The program runs in about one sixth the time on a COMPAQ 386 machine. Methods of reducing computer time required are discussed in Chapter 6.

4.3 Single Component, Dual Repair State

The second example problem is an extension of the first, which demonstrates a capability of the REPAIR.SUPERVISOR routine (a subroutine in the DYMCAK program that determines when component repair is initiated) and the ease at which the DYMCAK program can be modified to meet specific applications. The problem involves a single three state component which has an exponentially distributed repair delay time before the component begins the repair process.

In Appendix B the entire program listing is shown. In the REPAIR.SUPERVISOR process routine, line 31 contains the WAIT command used to create the third component state. It has been modeled as a Weibull distributed variable, but by proper choice of the parameters, the Weibull distribution becomes an exponential distribution. The Weibull distribution is characterized by the equation:

$$f_t = \left[\frac{a}{b} \right] \left[\frac{t}{b} \right]^{a-1} \exp \left[-\left[\frac{t}{b} \right]^a \right] \quad (4.3)$$

where, a and b are the distribution parameters. By letting the parameter a equal 1.0, the Weibull distribution becomes an exponential distribution with lambda equal to 1/b. Lines 23 and 24 define the exponential distribution with a mean failure rate of one failure every 100 hours. If, in the future, it is desireable to enter different delay distributions for various components, the parameters for the Weibull distribution can be read in the INPUT routine in the same manner as the repair distribution parameters.

The failure and repair rates for this example were chosen the same as for example number one. Thus with a mean repair delay time of 100 hours, the component now has three equal transfer rates from its three states. Thus it is evident that for the asymptotic case, the component will spend equal time in each of the three states. The component is only available when it is in its operational state, thus the asymptotic unavailability is two thirds.

To test the asymptotic unavailability the DYMCA program was run for a simulated component operation of 10,000 hours and 100 Monte Carlo trials. As in example one, the component was modeled as a passive element, although results would be the same for modeling the component as any of the other four elements for this simple case. Again the unavailability was sampled at 500 hour intervals to show the fluctuation of the value around the expected value of 0.6667 corresponding to two thirds. Table 4.2 shows the results which indicate again that for the small number of Monte Carlo trials used, the variance is quite large.

For this test the average system unavailability was also printed out for each of the 100 Monte Carlo trials. The range of values was divided into nine bins and the number of trials in each bin plotted against the central unavailability value for that bin. The results are shown in Figure 4.4. By using the fourth order Runge-Kutta technique to determine the time dependent

component unavailability over the 10,000 hour period, as is done for Figure 4.5, and then numerically integrating the result from time zero to 10,000 hours and dividing by the total time (10,000 hours), the exact result is found to be 0.6634. This indicates that the first 200 hours of operation did contribute slightly to lowering the result obtained. The simulation result agrees with the exact result within less than one percent difference. Again the standard deviation of the simulation result is ± 0.05 which may be insignificant for some analyses.

Table 4.2

Single Component, Dual Repair State Instantaneous Unavailability

| TIME | UNAVAILABILITY |
|---------|----------------|
| 0.0 | 0.0 |
| 500.0 | 0.63 |
| 1000.0 | 0.70 |
| 1500.0 | 0.70 |
| 2000.0 | 0.68 |
| 2500.0 | 0.67 |
| 3000.0 | 0.65 |
| 3500.0 | 0.67 |
| 4000.0 | 0.72 |
| 4500.0 | 0.69 |
| 5000.0 | 0.64 |
| 5500.0 | 0.59 |
| 6000.0 | 0.63 |
| 6500.0 | 0.68 |
| 7000.0 | 0.65 |
| 7500.0 | 0.68 |
| 8000.0 | 0.69 |
| 8500.0 | 0.68 |
| 9000.0 | 0.61 |
| 9500.0 | 0.70 |
| 10000.0 | 0.64 |

To compute the time dependent unavailability of this component the simulation time was reduced to 200 hours, and the number of trials increased to 1000 to reduce the variance of the results. Unavailability samples were taken every simulated hour and the results are plotted in Figure 4.5. For

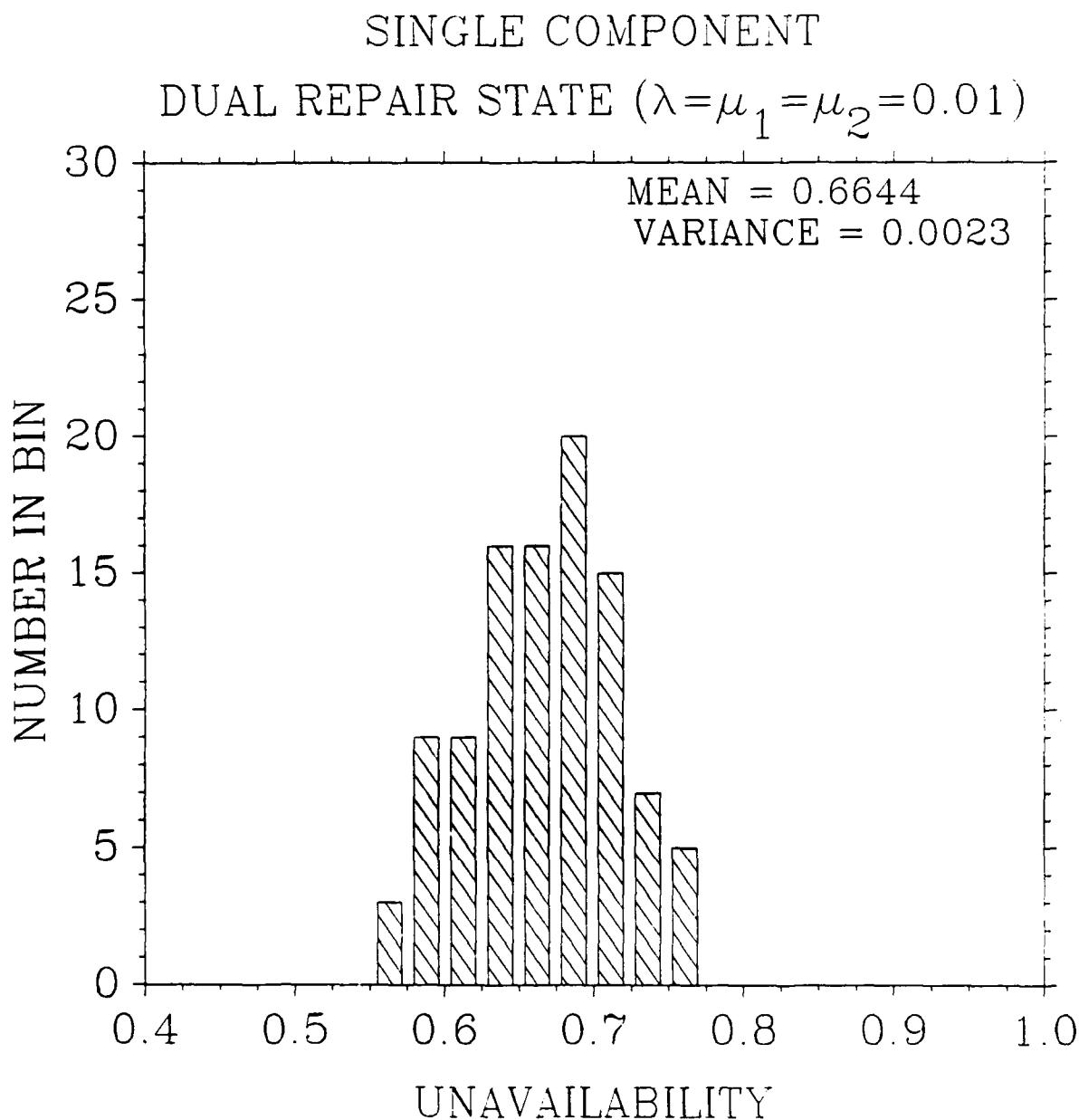


Figure 4.4 Single Component, Dual Repair State
Average Unavailability

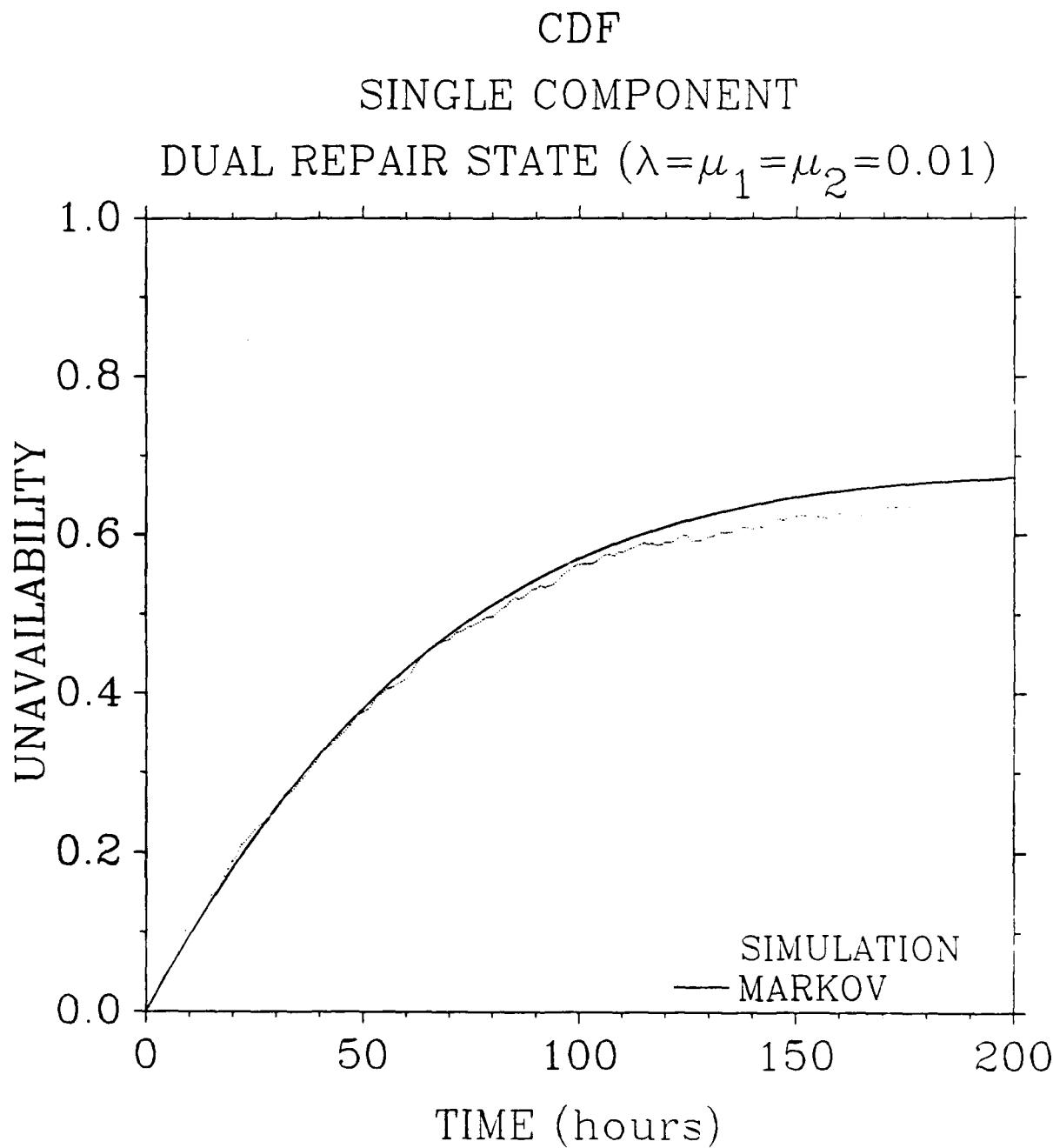


Figure 4.5 Single Component, Dual Repair State
Time Dependent Unavailability

in this example it is also possible to derive the analytic equations for the probability that the system is in any one of its three states using simple Markov procedures. The three equations are:

$$\frac{dP_0}{dt} = -\lambda P_0 + \mu_2 P_2, \quad (4.4)$$

$$\frac{dP_1}{dt} = -\mu_1 P_1 + \lambda P_0, \text{ and} \quad (4.5)$$

$$\frac{dP_2}{dt} = -\mu_2 P_2 + \mu_1 P_1, \quad \text{where } \lambda = \mu_1 = \mu_2 = 0.01 \quad (4.6)$$

Rather than solve these equations using Laplace transforms or matrix exponentiation techniques, a fourth order Runge-Kutta numerical integration routine taken from reference P-4 was used. The time dependent probability of the component being in state 0 was calculated and the result was subtracted from one to give the component unavailability. This result is plotted in Figure 4.5 for comparison with the simulation results.

From Figure 4.5 it is seen that the simulation program again gives good results for the time dependent unavailability. As the value of simulated time increases there is a fluctuation of the simulation results about the desired value, but as explained before this can be reduced by increasing the number of trials. The computer time required for these two experiments was comparable with the first example problem (approximately 30 minutes). The addition of the third component state did not significantly alter the time required to complete the run. The most important contributions to running time appear to be the length of simulation time for each trial and the number of time samples taken during each trial (the sampling process interrupts the simulation).

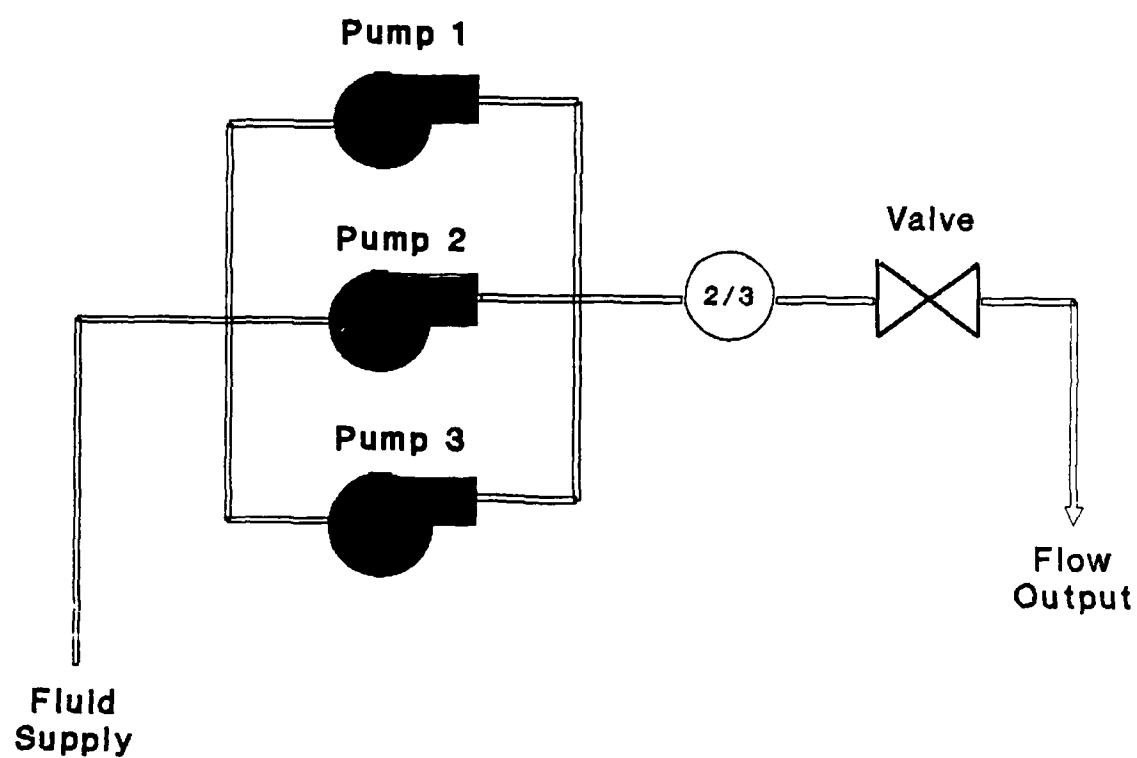
4.4 Two Out of Three Pumps

The third test case considers a more complicated system composed of

three pumps connected in parallel. Figure 4.6 shows a diagram of the system. The output of the pumps is fed to a common header where the flow then enters a valve. Success of the system requires at least two pumps to be operating and there to be flow output from the valve. The DYMCA program, as written, does not currently treat the strength of signals between components, thus a slight modification was required to allow this test since the program would not know if the output signal from the valve was the result of one, two, or three pumps operating. The modification also allows the determination of the system status simply by checking the output process signal from the valve.

The alteration made to the program for this test was made in line number 129 of the VALVE routine. By changing the test to require two input processes, the valve would not have an output unless at least two of the pumps are providing input to the valve. There are other ways this problem could have been modeled, but this method appeared to be the most straightforward.

Unlike the previous two examples, this problem does not have a simple analytic solution for the time dependent unavailability. To simplify understanding of the test results, all pumps are chosen to be identical and the valve was modeled with failure and repair distributions identical to the three pumps. There are four components which can be in either a failed or operational state which means the system can be in $2^4 = 16$ possible states. Since all failure and repair rates are equal, in the asymptotic case each system state has equal probability of occurrence. Only four of the states correspond to the system being in an available condition, thus twelve states (or three fourths of the states) contribute to system unavailability. Clearly the asymptotic unavailability should be 0.75.

Two Out of Three Pumps**Figure 4.6 Two Out of Three Pumps System Diagram**

As in the previous two examples, the program was run for a simulated time period of 10,000 hours and for 100 Monte Carlo trials. Table 4.3 shows the fluctuation of unavailability about the desired value of 0.75. Again, the failure and repair distributions were chosen to be exponential with mean values of 100 hours. Figure 4.6 indicates that the system has reached its asymptotic state after approximately 200 hours. Thus the actual value for average system unavailability should be slightly less than the value of 0.75, which would be the exact result achieved for average system unavailability as time goes to infinity. This was also the case with the first two examples.

Table 4.3
Two Out Of Three Component Instantaneous Unavailability

| TIME | UNAVAILABILITY |
|---------|----------------|
| 0.0 | 0.0 |
| 500.0 | 0.72 |
| 1000.0 | 0.80 |
| 1500.0 | 0.76 |
| 2000.0 | 0.75 |
| 2500.0 | 0.78 |
| 3000.0 | 0.78 |
| 3500.0 | 0.73 |
| 4000.0 | 0.74 |
| 4500.0 | 0.66 |
| 5000.0 | 0.76 |
| 5500.0 | 0.68 |
| 6000.0 | 0.66 |
| 6500.0 | 0.77 |
| 7000.0 | 0.78 |
| 7500.0 | 0.71 |
| 8000.0 | 0.73 |
| 8500.0 | 0.67 |
| 9000.0 | 0.77 |
| 9500.0 | 0.71 |
| 10000.0 | 0.81 |

The average value of unavailability over the 10,000 hour simulation was printed for each of the 100 trials and the resulting distribution is plotted in Figure 4.7. This figure indicates that the mean value of unavailability

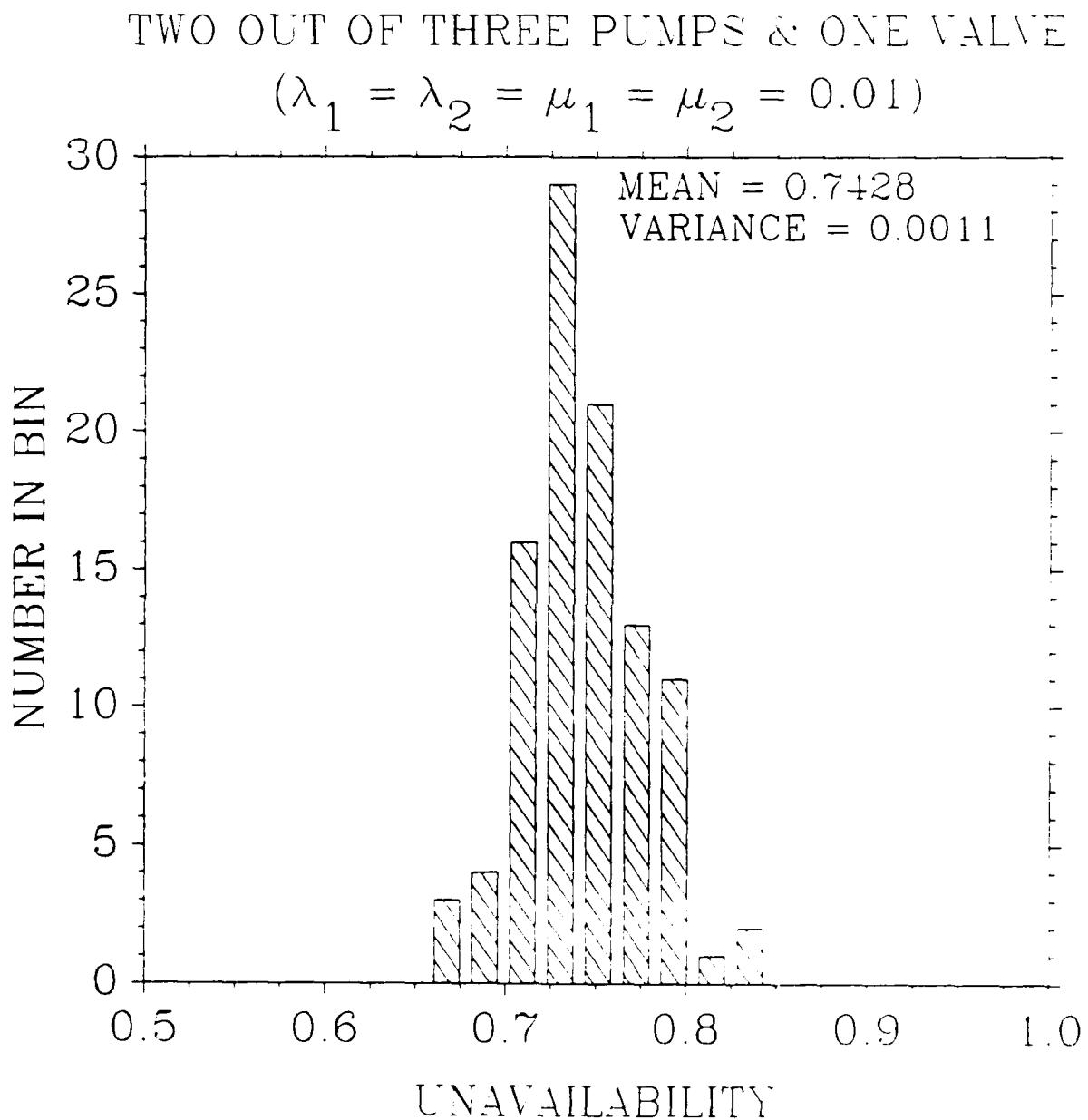


Figure 4.7 Two Out Of Three Component
Average Unavailability

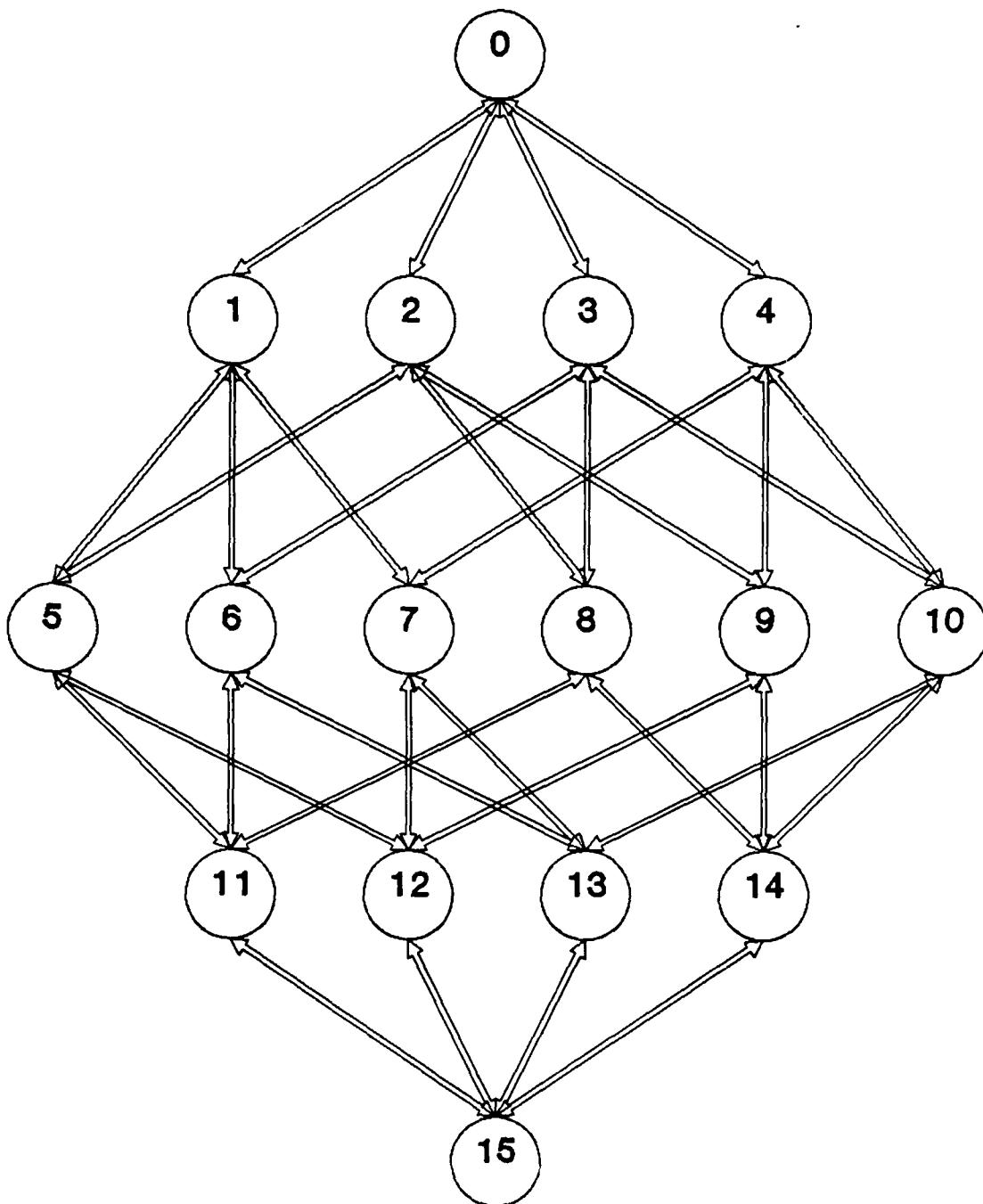
is 0.7428 which agrees well with the expected value of slightly less than 0.75, and it is seen that the standard deviation of the distribution is 0.03. This value of standard deviation is approximately half of that obtained for the first two example problems.

The time dependent performance of this system is also of importance, thus a second run was done over a simulated time period of 200 hours using 1000 Monte Carlo trials. The unavailability was sampled every hour to provide an accurate picture of the simulation program performance. For comparison, the Markov equations for the system were written. There are sixteen possible states and these are:

| | | |
|----|---|----------------------------------|
| 0 | - | All components are good |
| 1 | - | Pump #1 failed |
| 2 | - | Pump #2 failed |
| 3 | - | Pump #3 failed |
| 4 | - | Valve failed |
| 5 | - | Pumps #1 and #2 failed |
| 6 | - | Pumps #1 and #3 failed |
| 7 | - | Pump #1 and Valve failed |
| 8 | - | Pumps #2 and #3 failed |
| 9 | - | Pump #2 and Valve failed |
| 10 | - | Pump #3 and Valve failed |
| 11 | - | Pumps #1, #2, and #3 failed |
| 12 | - | Pumps #1 and #2 and Valve failed |
| 13 | - | Pumps #1 and #3 and Valve failed |
| 14 | - | Pumps #2 and #3 and Valve failed |
| 15 | - | All Components are failed |

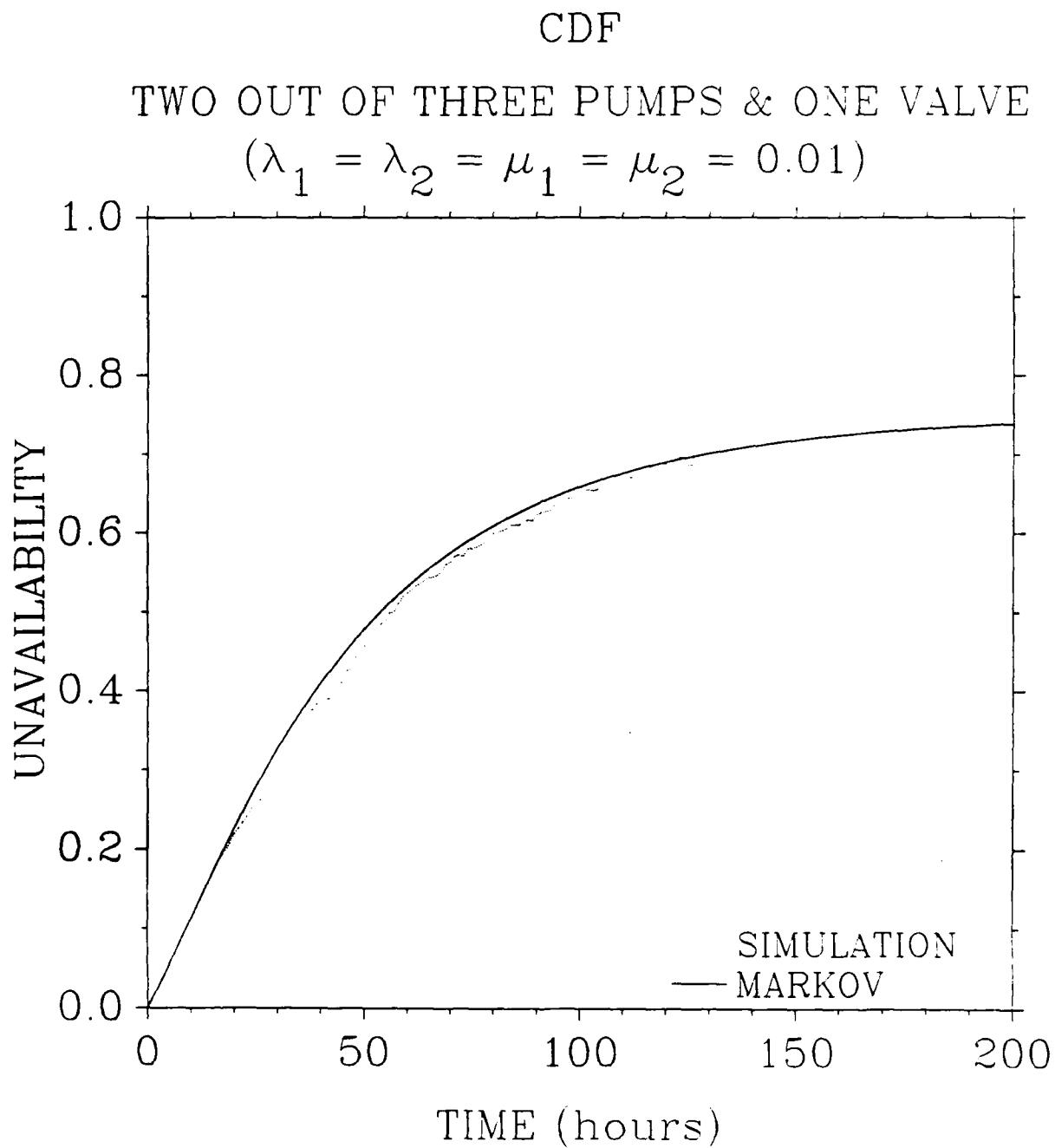
Figure 4.8 shows the Markov state transition diagram for this system. All transition time distributions are the same and given by $\lambda_1 = \lambda_2 = \mu_1 = \mu_2 = 0.01$ per hour.

Using these sixteen states the Markov equations for the system were written. These equations lead to a sixteen by sixteen matrix which is not a trivial problem to solve, therefore a fourth order Runge-Kutta numerical integration routine (from ref. P-4) was used to solve for the probability that the system is in any one of its sixteen states during the time interval



Note: All Transfer Rates are Equal

**Figure 4.8 Markov State Transition Diagram
For Two Out of Three Pump System**



**Figure 4.9 Two Out Of Three Component
Time Dependent Unavailability**

from zero to 200 hours. States 0, 1, 2, and 3 correspond to the system being available therefore the probabilities that the system is in any one of these four states is summed and subtracted from one to give the system time dependent unavailability. This exact solution is plotted in Figure 4.9 along with the simulation results for comparison.

It is seen from Figure 4.9 that even for this more complicated system, the DYMCAm simulation program provides good results for the system time dependent unavailability. Again the fluctuation of the results about the desired result can be seen at larger time values and it is evident that the accuracy of Monte Carlo analysis is directly related to the number of trials performed.

For this example problem, the computer time required to run the 10,000 hour simulation run for estimation of the asymptotic unavailability value was approximately three hours on an IBM compatible XT running at 7.16 MHz. The second run to determine time dependent unavailability required four and one half hours. The significant increase over the time required for the first two tests is due to the fact that this problem is more complicated (sixteen system states as apposed to two or three) which leads to a far greater number of calculations to be performed during execution of the program. The difference between the two times required for the asymptotic run and the time dependent analysis run reflects on the fact that for more complicated systems, the number of Monte Carlo trials performed will have the controlling effect on the amount of time required to complete a computer run. Considering the fact that the accuracy of the results is dependent on the number of trials performed, it is evident that methods should be explored to reduce the amount of time required for a computer run. The DYMCAm code could probably be programmed more efficiently. It was written to be as transparent

as possible and therefore may not be as efficient as possible.

4.5 GO-FLOW Example Problem

The fourth example problem considered will demonstrate the phased mission capability of the DYMCA program. For comparison, this problem is derived from the GO-FLOW example problem discussed in reference M-2. The solution derived from the methods of reference M-2 will be used for comparison with the results of the simulation method.

The problem to be solved involves a simple electrical circuit. Figure 4.10 gives a diagram of the system. It is composed of a battery, having a demand failure probability of 0.1, which will supply power to two parallel circuits. Each circuit has a switch and a light bulb. The switches are identical and have a demand failure probability of 0.3. Neither the battery nor the switches are presumed to experience run time failures. The light bulbs in the system are considered identical and they have a 0.2 probability of failing on demand and an exponentially distributed run time failure rate with a mean value of one failure every 1,000 hours.

The actual problem solved in reference M-2 considered that the switches had a probability of premature closure, however in the DYMCA model this type of failure would be modeled as a run time failure and would mean that there is an equal probability that the switch could open once it is closed. Since the latter condition was not considered in reference M-2, the premature failure probability was excluded from the simulation analysis. A numerical solution was performed on the modified GO-FLOW problem to provide the comparison results.

The phased mission problem to be solved considers that at time zero the battery is connected to the circuit and has a 0.9 probability of being good. A fraction of a second later one of the switches is closed, then ten hours

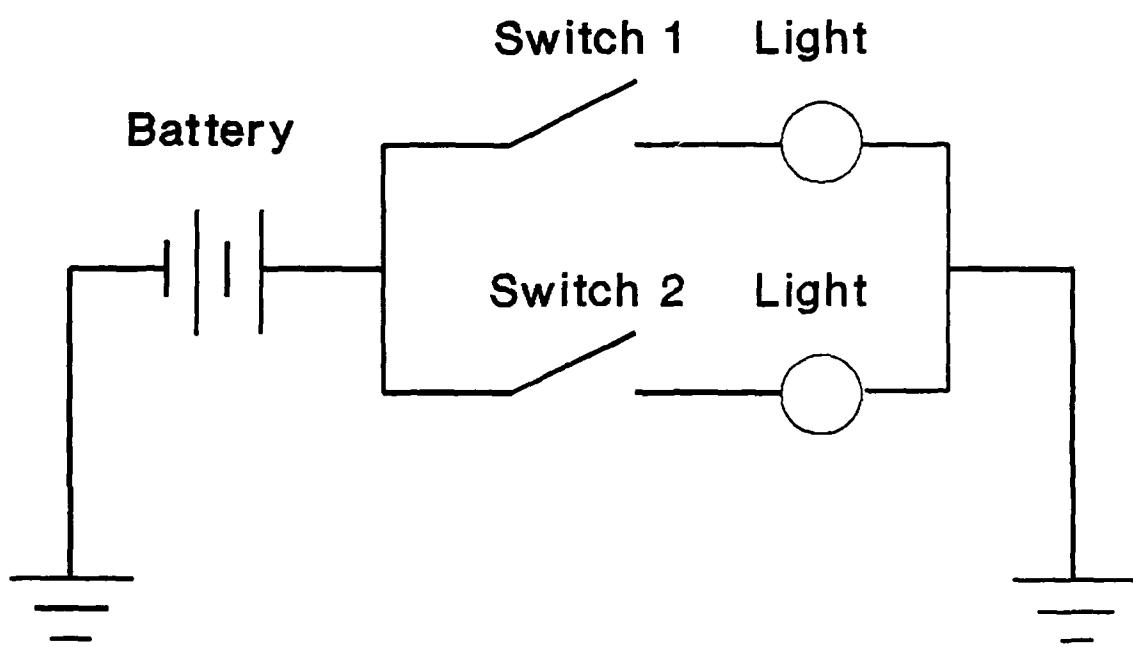


Figure 4.10 Light Bulb Problem Diagram

later the second switch is closed. The analyst wishes to determine the probability that at least one light is on immediately following closure of the first switch (call this time $t=0.0$), immediately prior to closing of the second switch (time $t=9.99$ hours), instantly following closure of the second switch (time $t=10.0$), and twenty hours after closure of the first switch (time $t=20.0$). Analysis using the DYMCA program was done varying the number of Monte Carlo trials from 1,000 to 10,000 to demonstrate a sensitivity analysis of the simulation method.

To solve this problem using the DYMCA program, the external event feature was used. This capability allows the input file to contain instructions which will cause a signal to change at an instant of time after the start of the simulation. This function was used to give the battery a process signal input at time $t=0.0$, to give the first switch a command signal to close at time $t=0.0$, and to give the second switch a command to close at time $t=10.0$ hours. This unique feature allows the DYMCA program to easily solved phased mission problems.

Tables 4.4 and 4.5 summarize the results of the ten tests run on the DYMCA program. Table 4.4 shows the results using from 1,000 to 5,000 Monte Carlo trials and Table 4.5 shows the outcome of tests using 6,000 to 10,000 trials. The tables show the actual probability of at least one light being on at each of the four designated time points as calculated using the GO-FLOW method and the corresponding values calculated with the simulation program. The difference of the simulation value from the actual value is shown and the percent error is calculated as the difference divided by the actual value. For an indication of the variance, the number of trials which would need to have been changed to give the actual results are indicated. For example, for the time point $t=20$ hours and for 1,000 trials, Table 4.4 indicates that

-10 trials would have to be changed. This means that 10 of the 1,000 trials for which a light was not on at t=20 would need to have had a light test on in order for the simulation results to agree with analytic results.

Table 4.4**Light Bulb Problem Results (1,000 to 5,000 trials)**

| QUANTITY | ACTUAL | NUMBER OF TRIALS | | | | |
|------------------------------|--------|------------------|--------|--------|---------|---------|
| | | 1000 | 2000 | 3000 | 4000 | 5000 |
| <u>TIME 0.0 hours</u> | | | | | | |
| Result | 0.5040 | 0.4910 | 0.5070 | 0.5057 | 0.5033 | 0.5020 |
| Difference from actual value | ---- | -0.0130 | 0.0030 | 0.0017 | -0.0007 | -0.0020 |
| Equivalent number of trials | ---- | -13 | 6 | 5 | -3 | -10 |
| Percent Error | ---- | -2.6 | 0.6 | 0.3 | -0.1 | -0.4 |
| <u>TIME 9.99 hours</u> | | | | | | |
| Result | 0.4990 | 0.4880 | 0.5025 | 0.5010 | 0.4985 | 0.4968 |
| Difference from actual value | ---- | -0.0110 | 0.0035 | 0.0020 | -0.0005 | -0.0022 |
| Equivalent number of trials | ---- | -11 | 7 | 6 | -2 | -11 |
| Percent Error | ---- | -2.2 | 0.7 | 0.4 | -0.1 | -0.4 |
| <u>TIME 10.0 hours</u> | | | | | | |
| Result | 0.7236 | 0.7060 | 0.7270 | 0.7320 | 0.7275 | 0.7266 |
| Difference from actual value | ---- | -0.0176 | 0.0034 | 0.0084 | 0.0039 | 0.0030 |
| Equivalent number of trials | ---- | -18 | 7 | 25 | 16 | 15 |
| Percent Error | ---- | -2.4 | 0.5 | 1.2 | 0.5 | 0.4 |
| <u>TIME 20.0 hours</u> | | | | | | |
| Result | 0.7191 | 0.6980 | 0.7205 | 0.7257 | 0.7215 | 0.7212 |
| Difference from actual value | ---- | -0.0211 | 0.0014 | 0.0066 | 0.0024 | 0.0021 |
| Equivalent number of trials | ---- | -21 | 3 | 20 | 10 | 10 |
| Percent Error | ---- | -2.9 | 0.2 | 0.9 | 0.3 | 0.3 |

Table 4.5

Light Bulb Problem Results (6,000 to 10,000 trials)

| QUANTITY | ACTUAL | NUMBER OF TRIALS | | | | |
|------------------------------|--------|------------------|---------|---------|---------|---------|
| | | 6000 | 7000 | 8000 | 9000 | 10000 |
| <u>TIME 0.0 hours</u> | | | | | | |
| Result | 0.5040 | 0.4995 | 0.4950 | 0.4971 | 0.4998 | 0.5007 |
| Difference from actual value | ---- | -0.0045 | -0.0090 | -0.0069 | -0.0042 | -0.0033 |
| Equivalent number of trials | ---- | -27 | -63 | -55 | -38 | -33 |
| Percent Error | ---- | -0.9 | -1.8 | -1.4 | -0.8 | -0.7 |
| <u>TIME 9.99 hours</u> | | | | | | |
| Result | 0.4990 | 0.4935 | 0.4889 | 0.4913 | 0.4939 | 0.4948 |
| Difference from actual value | ---- | -0.0055 | -0.0101 | -0.0077 | -0.0051 | -0.0042 |
| Equivalent number of trials | ---- | -33 | -71 | -62 | -46 | -42 |
| Percent Error | ---- | -1.1 | -2.0 | -1.5 | -1.0 | -0.8 |
| <u>TIME 10.0 hours</u> | | | | | | |
| Result | 0.7236 | 0.7238 | 0.7204 | 0.7205 | 0.7214 | 0.7243 |
| Difference from actual value | ---- | 0.0002 | -0.0032 | -0.0031 | -0.0022 | 0.0007 |
| Equivalent number of trials | ---- | 1 | -22 | -25 | -20 | 7 |
| Percent Error | ---- | 0.03 | -0.4 | -0.4 | -0.3 | 0.1 |
| <u>TIME 20.0 hours</u> | | | | | | |
| Result | 0.7191 | 0.7185 | 0.7143 | 0.7145 | 0.7154 | 0.7186 |
| Difference from actual value | ---- | -0.0006 | -0.0048 | -0.0046 | -0.0037 | -0.0005 |
| Equivalent number of trials | ---- | -4 | -34 | -37 | -33 | -5 |
| Percent Error | ---- | -0.1 | -0.7 | -0.6 | -0.5 | -0.1 |

It can be seen in these two tables that the error decreases as the number of trials is increased and for 10,000 trials the percent difference between the actual availability values and the estimates from the simulation program are less than one percent for all time points. As expected, there

is very little difference in the error percentages for two cases separated by only 1,000 trials. For example, there is an average of only a 0.5 percent difference between the values for the 3,000 trial case and the 4,000 trial case. The amount of error should decrease with increasing number of trials in proportion to one over the square root of the number of trials and this is evident by comparing the 1,000 and 10,000 trial cases. If the values of percent error at 1,000 can be taken to be characteristic of values that would be obtained regardless of the random number generator used by the program, then certain comments about the error can be made. For the four time points, the average error for 1,000 Monte Carlo trials is approximately 2.5 percent. The variance of the estimates should go as one over the square root of the number of trials, therefore it is expected that the percent error for 1,000 Monte Carlo trials should be no greater than the square root of 1,000 divided by the square root of 10,000 times the error for the 1,000 trial case. Checking this assumption it is seen that the expected error should be no greater than 0.8 percent. From table 4.5 it is evident that this assumption is indeed correct and it seems probable that for any number of trials used greater than 10,000 the percent error of the result should be no greater than 0.8 percent. However, to significantly reduce the error below this 0.8 percent value using the current program would take a prohibitively large number of trials. In fact, using the above methods it is estimated that to reduce the error margin to 0.5 percent or less would require in excess of 25,000 trials.

The computer time required for these tests was approximately fifty minutes for every 1,000 trials, thus the 10,000 trial case took about eight and one half hours to run. This time requirement refers to an IBM compatible XT running at 7.16 MHz. The approximate time for the 10,000 trial on a 386

personal computer is estimated to be about 1.5 hours. Even at this rate, though, it would take computer runs of close to five hours to reduce the error to less than 0.5 percent for this particular problem with the current structure of the DYMCA program. If more accurate solutions were necessary, it is clear that modifications to the program will be essential on order to reduce the computer time requirement, however, this kind of accuracy is almost never needed since there is always a inevitable amount of uncertainty in the original data.

Demonstrated by this example was the important DYMCA feature of using external events in phased mission analysis problems. GO-FLOW was designed with this capability, but most other methods do not provide an easy method for treating this type of problem. This capability can be exploited to analyze all types of systems involving testing and maintenance functions.

4.6 Chapter Summary

In this chapter four example problems have been analyzed using the DYMCA dynamic simulation model. A single component with exponential repair and failure distributions was considered to demonstrate program operation. Next, a third state was added to the component in the form of an exponentially distributed delay time between the failure and repair states. This example demonstrated use of the REPAIR.SUPERVISOR routine and the ease with which the program can be modified to meet specific needs. The third example considered a more complex problem and demonstrated that the program can model m-out-of-n components, although to do this properly the program should be modified to consider the strength of process variables. The final example treated a simple phased mission problem and illustrated the use of the external event concept to turn a component on after the start of a simulation time period. The results of all four examples were compared with

analytic answers and the comparison was favorable. In each case the simulation values agreed with expected results quite well.

It was seen that the simulation method can be used to evaluate the asymptotic unavailability of a system, but more importantly, that it also provides good results for a time dependent unavailability analysis. Analytic asymptotic values agreed almost exactly with mean values of unavailability distributions produced. Time dependent simulation results agreed well with Markov solutions; however, differences between simulation and exact results do not vanish as the simulation proceeds.

The DYMCA program can be used for any type of phased mission problem where it is necessary to turn components on and off during a simulated time period. This capability was demonstrated with a very basic problem. This potential should prove very powerful in systems reliability analysis. Most current techniques are not designed for phased mission analysis.

An important result observed was the importance of the program result accuracy on the number of Monte Carlo trials used and the time requirement necessary to achieve satisfactory results. A simulation technique such as this provides an estimate of the unavailability of a system. This estimate will have a distribution associated with it. The mean of the distribution should equal the exact analytical result, if one is obtainable, and the variance of the distribution is related to the number of trials performed. Thus, though the mean of the distribution may be equal to the exact solution after a small number of trials, there is no way of knowing this for certain unless the analytical solution is available. The variance of the distribution provides a measure of "confidence" in the mean value, thus to have increased confidence in the simulation result, it is desireable to have small variances. To accomplish this may require large amounts of computer

time.

The DYMCA M program shows that it provides accurate results for simple problems. Further work should be done to modify the program to handle different levels of process signals and also to improve the speed with which the program runs. In the next chapter a problem involving a continuous process will be treated which involves modifying the program extensively and demonstrates the capability of simulation programs to treat continuously variable signals.

Chapter 5

Continuous Simulation TANK Program

5.1 Introduction

Most reliability analysis methods are designed to treat only systems which can be modeled using a discrete state space. This type of approach, however, may not be adequate for analyzing certain systems including process control problems which depend on continuous variables such as pressure, temperature, flow rates, and tank water levels. This particular type of problem has been discussed in detail by Aldemir in references A-1 and A-2. Aldemir has developed a dynamic method which uses discrete Markov chains to model the probabilistic behavior of the system to analyze such problems, and in references A-1 and A-2 he applies his technique to several examples including a process control system which regulates the water level in a tank.

The DYMCA dynamic simulation model discussed in previous chapters does not have the capability to treat failures of components whose state depends on a continuous variable such as water level. In this chapter the basic program is modified to include this capability. Although the TANK program developed here is designed specifically to solve the example problem treated in references A-1 and A-2, it demonstrates the capability of the basic simulation approach to handle analysis of complex systems which involve continuous variables.

The chapter begins with a section describing the problem to be solved. This problem is similar to the example treated by Aldemir in reference A-1. Aldemir treats the problem by considering the failure of the control system itself, which means that if a unit is in a failed state in one system control region, this does not mean the component will remain in that state if the process variable moves to another control region. The problem treated here

considers only the failure of individual input and output units. Thus once a component fails, it remains in that failed state regardless of any changes that may occur in the operating state of the system. Thus results should be different than those predicted by Aldemir. Once the problem is described, program modifications will be indicated which were necessary to simulate the example. As much as is possible the DYMCA program is left exactly as it was in previous chapters and special subroutines and processes are added to treat the continuous variable.

After the TANK program development is explained, the procedure is used to solve two of the cases discussed by Aldemir for the process control problem which controls a tank water level. Results are compared qualitatively with results in the reference. A simple Markov approximation to the system is also developed and results of the TANK simulation program are compared quantitatively to this solution method. For the simple case tested, results of the simulation model agree well with the approximate Markov modeling of the system and also with Aldemir's solution. For the more complicated case tested, simulation results agree with Markov approximations but not with results proposed by reference A-1. Explanations are given for the difference.

The TANK simulation program performs well on the specific problem tested and demonstrates the capability of a Monte Carlo simulation approach to be used in solving a continuous variable problem.

5.2 Problem Description

The problem to be solved consists of a fluid containing tank which has three separate level control units. Figure 5.1 shows a diagram of the system. Each control unit is independent of the others and has a separate level sensor associated with it. The level sensors measure the fluid level

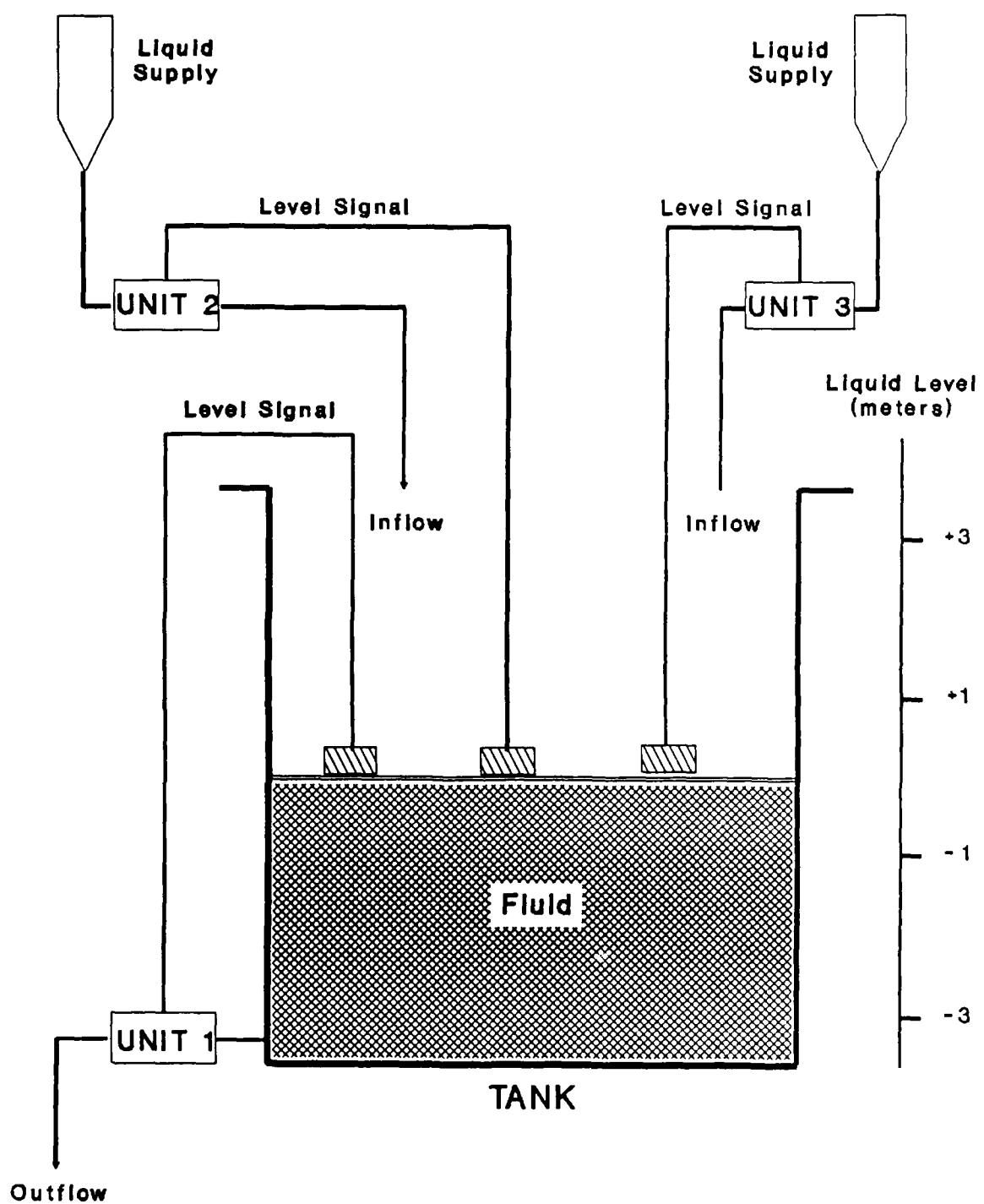


Figure 5.1 Tank Problem Diagram

in the tank, which is a continuous process variable, and based on the information from the level sensors, the operational state of the control units is determined. Each flow control unit can be thought of as containing a controller which turns the unit on and off based on the signal from the level sensors, as shown in Figure 5.1. Failure of the system occurs when the tank either runs dry or overflows.

The tank has a nominal fluid level at the start of system operation of zero meters. The maximum level of the tank is 3 meters (point b) and the minimum level of the tank is -3 meters (point a). If the tank level moves out of this range, failure of the system has occurred. Within this range there are two set points at -1 meter (set point alphal) and +1 meter (set point alpha2). These set points define three control regions for system operation. Region one is defined from point a to alphal, region two is from alphal to alpha2, and region three is from alpha2 to point b. When the fluid level is in any of the three control region there is a specific action required of each of the three control units. Each control unit acts independently and is not aware of what the state of the other control units is except through the change occurring in the process variable. Table 5.1 shows the control unit states for each control region.

Table 5.1

Flow Control Unit States as a Function of Fluid Level

| Control Region | Liquid Level (x) | Control Unit 1 | Control Unit 2 | Control Unit 3 |
|----------------|---|----------------|----------------|----------------|
| 1 | $x \leq \text{alphal}$ | off | on | on |
| 2 | $\text{alphal} \leq x \leq \text{alpha2}$ | on | on | off |
| 3 | $\text{alpha2} \leq x$ | on | off | off |

Unit one is an outlet element providing a means for releasing fluid

from the tank to lower the level. In all cases discussed in reference A-1, unit one is assigned an exponential failure distribution with a mean failure time of 320 hours. This is the failure rate of unit one transferring to the wrong state. When operating the unit allows fluid to flow out of the tank at the rate of 0.01 meters per minute. Unit one receives a command from the level controller to be on when the fluid level is in control region two or three and it receives a signal to shut off when the fluid level is in control region one. If the unit is modeled as a valve, it is clear that the valve is normally open unless the fluid level is below the low level setting for the tank, in which case the valve is closed. The component routine used to model unit one as a valve is one of the routines contained in the basic DYMCA program code.

Unit two is a supply unit which provides fluid input to the tank. It too has an exponentially distributed failure rate which applies to unit two transferring to the wrong state. The mean failure time used for all cases considered in reference A-1 is 219 hours. When operating, the unit supplies fluid at the rate of 0.01 meters per minute. This unit receives a control signal to be on if the fluid level is in control regions one or two, and it receives a signal to shut off if the fluid level is in control region three. The unit can be modeled as a pump or an inlet valve. In this work the unit was treated as a valve. It is only necessary that the component model be able to fail open (on) or closed (off) and that it respond to control signals.

The third unit is also a fluid supply element. It is identical in nature to unit two except that it has a mean failure time of 175 hours. Through most of the cases treated in reference A-1, the flow rate from unit three is identical to unit two therefore providing 0.01 meters per minute of

fluid, however in one of the cases (Case F of Ref. A-1), which is also considered here, unit three only supplies 0.005 meters per minute of fluid. Unit three is normally in an off (closed) state unless the fluid level drops into control region number one, in which case the unit receives a signal to turn on. Like unit two this unit can also be modeled as an inlet valve as is done in the following analysis.

At the start of system operation the fluid level is in the normal region (control region two) and units one and two are on while unit three is off. Thus the flow rate into the tank is equal to the flow rate out of the tank, and the fluid level is not changing. This state will continue until one of the level control units fails. Then the fluid level will change either up or down depending on which unit has failed, and when the fluid level enters a new control region the controller will take action to halt the change. The new system state may or may not be stable, as is seen later in the chapter, however failure of the system cannot occur with the failure of a single control unit. The level will remain in the new control region, or oscillating between two control regions until a second unit fails. The second failure is likely to cause the system to fail by the tank either running dry or overflowing.

Since component repair is not considered in this problem, all scenarios will end in system failure. The type of failure experienced is dependent on the sequence in which the units fail and also upon the timing of failure for certain cases in which the fluid level oscillates. The problem to be solved in this reliability analysis is to determine the time dependent probability of each of the two types of failure. The complication which prohibits this type of problem from being easily solved by other analysis methods, is that component states are dependent on a continuous process variable. Modeling

of the process variable must be done, and a method must be available by which control units are allowed to change state at non-deterministic times. In other words the method of the DYMCA program, which uses external events to control phased mission problems, is not appropriate since the time at which a component will be required to change its operating state will not be known before the simulation is begun.

One characteristic of this problem does allow for a simple method of approximating the failure results. This is the relationship between unit failure time and the time required for the system to change from one control region to another once a unit has failed. The three units have mean failure times of 320, 219, and 175 hours respectively. If the tank fluid level is at zero when a unit fails, then at a flow rate of 0.01 meters per minute it will only take approximately 1.7 hours for the tank to change control regions. If the level is at the edge of control region one, and must travel to control region three, the longest amount of time that will be required is approximately 3.5 hours. If these times can be considered small enough so that the assumption can be made that a second failure does not occur until the system has entered a new control region, then a straightforward approach of initiating event analysis can be used and simple Markov chains can be applied to solve the problem. This approach is used as an approximate method against which the simulation results can be checked for an estimate of simulation performance.

For the second case treated, in which the flow rate from unit three is reduced to 0.005 meters per minute, there are failure sequences which will lead to the fluid level changing at only 0.005 meters per minute. In this case the maximum time required to change from one control region to another is approximately 7 hours. Clearly the assumption that a second failure does

not occur during this transit time period is not as good for this case, and results using the approximate technique will not be as accurate. However, results are still expected to be quite good.

5.3 The TANK Program - Modifications to DYMCAm

The major change which was necessary to make in the DYMCAm program in order to solve the tank problem was to add a routine which models the continuous process variable. SIMSCRIPT II.5 has a continuous variable modeling capability , which is described in reference F-1, and this was used to treat the fluid level in the tank. This new variable required the addition of several subroutines to the DYMCAm program and these are described in this section. In addition, certain subroutines of the original program required minor modification. Table 5.2 lists all the new subroutines added and all the old subroutines to which adjustments were made. A complete SIMSCRIPT program listing of the new subroutines is contained in Appendix C. The modified subroutines are contained in Appendix B. In Appendix B, those subroutines which were modified for the tank problem contain the message "TANK" at the far right hand side of the page next to the added or altered lines of code. These commands should be removed or altered to use the DYMCAm program by itself. It should be emphasized that the sole purpose of the particular modified program is to demonstrate a simulation modeling approach to a reliability problem involving continuous process variables. Modifications made to the DYMCAm program have been chosen with an eye on rapid implementation rather than programming generality.

The most fundamental addition to the program was the TANK process. This is the continuous process which provides SIMSCRIPT with the capability to solve continuous variable systems. The continuous capability of SIMSCRIPT II.5 is described in detail in reference F-1 by Fayek. The difference

between discrete and continuous event simulation is fundamental. In purely discrete event simulation the model advances in time from event to event using entries in an event queue. It is assumed that the system remains unchanged between scheduled events and can change only at the designated event times. For a continuous model, variables are assumed to vary continuously with advancing time. Thus time is incremented by a small amount and all variables are updated. This is done by associating a differential equation with each continuous variable which indicates the rate of change for that variable. Then as time is advanced by discrete time steps, integration is performed to update the status of the continuous variable at the end of each time step.

Table 5.2

TANK Subroutines

| Subroutine | Description |
|---|---|
| <u>Modified DYMCA M Routines</u> | |
| PREAMBLE | Modified to reflect new variables and processes |
| MAIN | Modified to Initialize and stop the Tank |
| CALL.UPDATE | Modified to start Tank process |
| RUN.INITIALIZE | Modified to add signals |
| SYSTEM.UPDATE | Modified to update flow rates |
| <u>New Routines</u> | |
| FLOW.UPDATE | Routine to calculate flow to and from Tank |
| STOP.TANK | Process to reset Tank after each trial |
| TANK | Continuous process to monitor fluid level |
| TANK.CONDITION | Function that checks for proper control region operation |
| TANK.INITIALIZE.RUN | Routine to initialize all variables and sets for the Tank |
| TANK.INITIALIZE.TRIAL | Routine to re-initialize specific variables for next trial |
| TANK.UPDATE | Routine to track System status and control all units |
| WATER.LEVEL | Routine providing integration quantity for continuous routine |

SIMSCRIPT II.5 uses a variable time step for which the user must specify the minimum and maximum to be allowed. The integration routine can be specified explicitly, or the Runge-Kutta integration routine which is contained in the SIMSCRIPT language may be used. Also associated with the integration routine are error parameters must be indicated to specify the accuracy of integration calculations desired. All of these initializations were entered in the TANK.INITIALIZE.RUN routine.

Figure 5.2 shows a flow chart of the operation of the TANK program. Following through this chart will provide an explanation of the TANK program operation and methodology. The function of routines of the DYMCA program will not be repeated here since they are described in Chapter 3.

The tank begins with the TANK.INITIALIZE.RUN routine which creates and initializes the variables and signals associated with the tank. This is done only once at the beginning of each computer run. Next, for every trial the tank output signals, the tank level, and the initial flow rate are reset by the TANK.INITIALIZE.TRIAL routine. After all other initialization is completed by the DYMCA program, the simulation clock is started. Failure of all three units will be scheduled to occur at discrete times in the simulation based on their failure rates, and these times are assigned by the DYMCA program.

Unlike the DYMCA program, which uses only discrete event simulation, the TANK program also contains the continuous tank level variable. Thus after the start of the simulation, control of the time aspect of the program is performed by the Tank process. This subroutine contains the statement (line 15):

work continuously evaluating 'water.level' testing 'tank.condition'

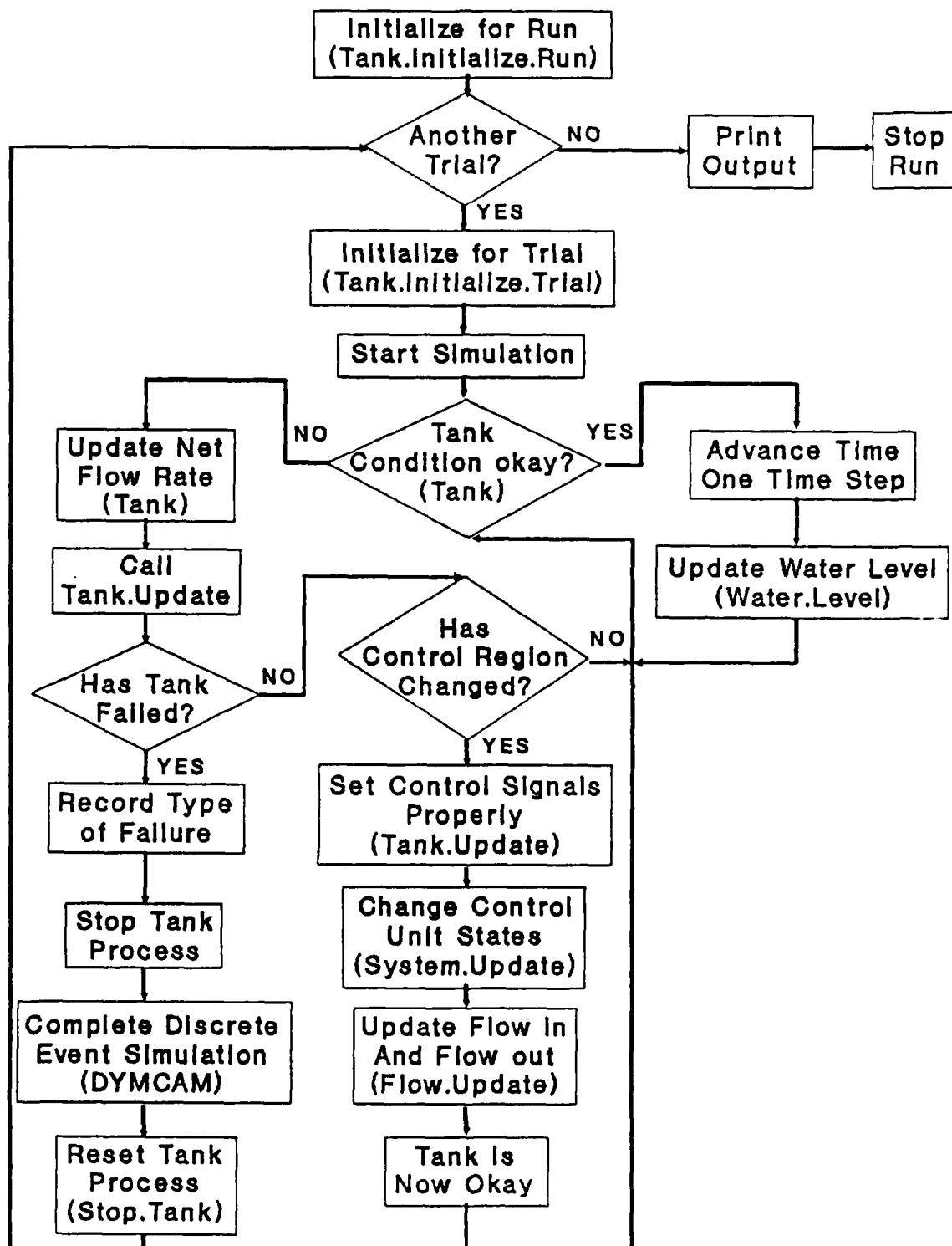


Figure 5.2 Flow Chart of TANK Program

This statement updates the tank water level using the WATER.LEVEL routine which applies the differential equation:

```
d.level(tank) = net.flow.rate(tank)
```

The time step is variable between the minimum and maximum specified by the user and in this case, is not variable since both values were set equal to one hour. If a variable time step were allowed, then SIMSCRIPT would adjust the step based on how fast the variable is changing. The integration routine, Runge-Kutta in this case, calculates the water level at the new time.

Once the new level is determined, the TANK.CONDITION routine is called to verify that the tank condition is good. If it is, then the simulation clock is advanced another time step, and the new water level is calculated. If the TANK.CONDITION routine determines that: 1) the net.flow.rate(tank) does not equal the flow.rate.in minus the flow.rate.out, 2) the tank has failed by overflow or dryout, or 3) The control state is not correct based on the current fluid level; then continuous time steps are stopped and control continues in the TANK process. The net flow rate for the tank is updated here next. The reason for this is to provide proper synchronization for changing of the flow rate. After updating the net flow rate, the TANK process calls the TANK.UPDATE routine.

The TANK.UPDATE routine serves two functions. First it checks the water level to see if overflow or dryout has occurred. If either condition has occurred, then the output signal from the tank, indicating tank status, is set equal to zero (representing tank failure), and control is returned to the Tank process. The TANK process then suspends itself. The rest of the simulation time of the trial passes in discrete event fashion. When the scheduled STOP.TANK and STOP.SIMULATION times are reached, the TANK process

is reset and the next trial begins. It should be noted that the system indicator variable can have only one of two values indicating either system success or failure. Since both tank overflow and tank dryout are failure events, it is necessary to simulate failure in each mode separately. This is done by altering the computer code to count only failures of one type or the other during a particular run of the program. To test for the probability of tank overflow, lines 13 through 17 of the TANK.UPDATE routine were rendered un-executable, and when testing for tank dryout, lines 13 to 17 were restored and lines 24 through 28 of the Tank.Update routine were removed. In either case, once the tank has run dry or overflowed, continuous operation of the system is suspended. Of course, an alternate modification is to revise the SYSTEM.UPDATE and RUN.OUTPUT routines such that multiple output states are recognized. This was felt to be more complex than the method adapted.

If the tank has not failed, then the TANK.UPDATE routine checks to see if the unit control states are correct based on the fluid level of the tank. If not, the TANK.UPDATE routine creates the proper control signals to send to the three units to change their operating state to the proper condition. To cause the units to change state, the SYSTEM.UPDATE routine is called. This is a DYMCAIM routine which changes the states of components based on changes in signals and on changes in other system component states. A new line added to the SYSTEM.UPDATE routine for the TANK problem, appears at line 141. This command causes the FLOW.UPDATE routine to be called. This routine calculates the flow rate going into the tank and the flow rate coming out of the tank based on the state of the three control units. It does not directly calculate the net flow rate into the tank which is used by the WATER.LEVEL routine. This is done in the TANK process to prevent the flow rate from

changing during an integration time step.

Once the flow rates are updated, control is returned to the SYSTEM.UPDATE routine. The SYSTEM.UPDATE routine, in turn, returns control to the Tank.Update routine. Now the tank is in the proper operating condition and thus control is returned to the TANK process. Since the tank has not yet overflowed or run dry, the TANK process begins execution of the continuous function again. Time is advanced by the given time step (one hour), the level of the tank is updated, and the condition of the tank is again checked. As long as the tank condition is good, operation continues in this fashion. If the tank condition tests bad, then the continuous operation is again suspended.

The failure rates used for the three control units in the tank problem make it highly likely that the system will fail during the simulated 1,000 hour time period, therefore at some point the continuous process should stop and the simulation will continue in the discrete event fashion. In the rare case of no system failure during the 1,000 hour period, the continuous process will be suspended by the Stop.Tank routine at the 1,000 hour time point, and the system will be reset for the next trial. Of course, no failure event would be recorded for such a trial.

Individual control unit failures are controlled by the DYMCA program. When a failure occurs, the SYSTEM.UPDATE routine is called which in turn will cause the flow rate into and out of the tank to be adjusted. This change will effect the TANK program when the TANK.CONDITION routine detects that the net flow rate to the tank does not equal the flow rate in minus the net flow rate out, and as described above, the continuous operation will be interrupted while the net flow rate is changed by the TANK process.

The new routines, TANK.INITIALIZE.RUN and TANK.INITIALIZE.TRIAL are

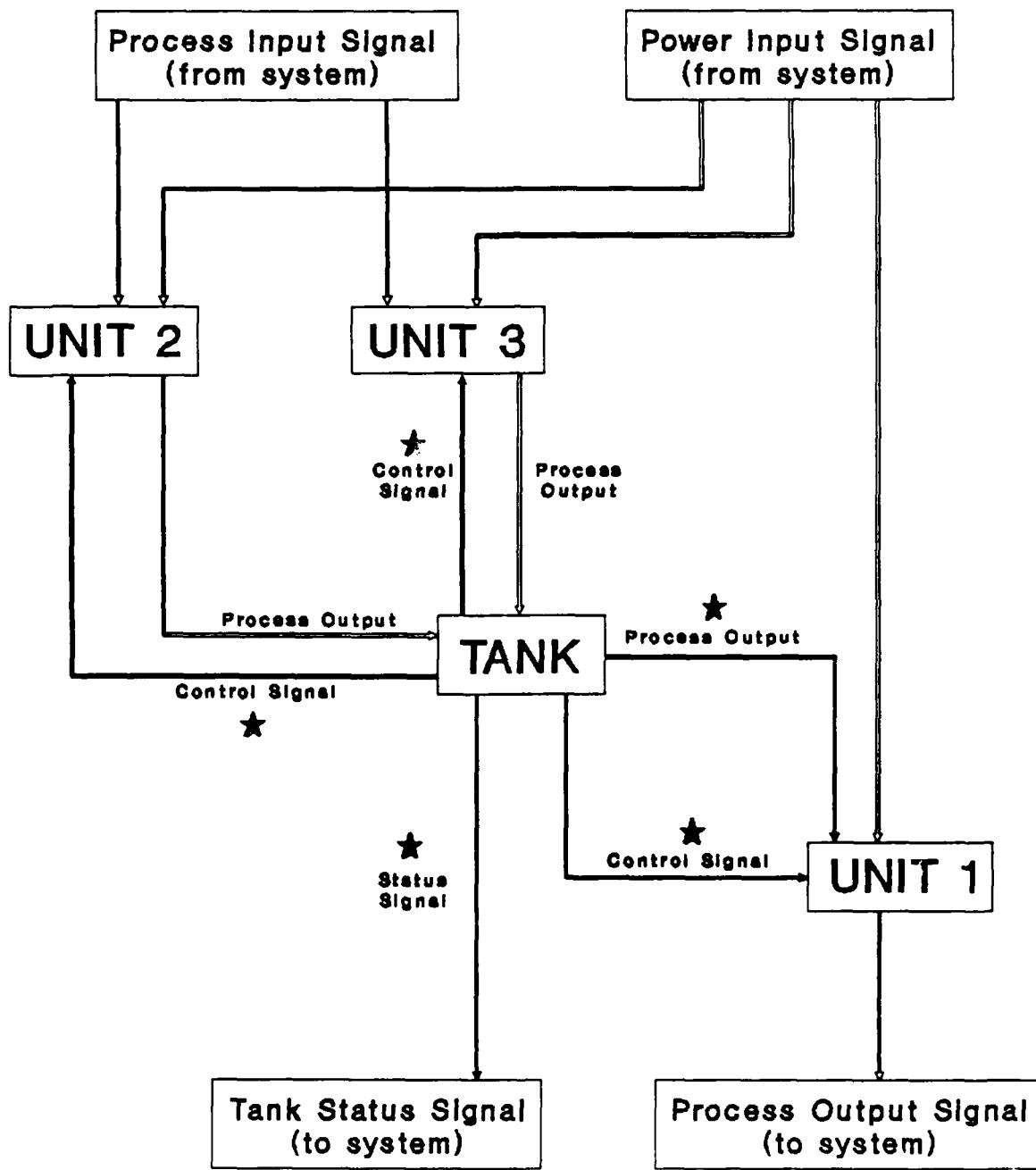


Figure 5.3 TANK Program Signals

used to initialize all the parameters associated with the test. Most importantly the TANK.INITIALIZE.RUN routine creates all of the output signals associated with the tank. Since the DYMCA program does not recognize the tank as being a component, it is not assigned any output signals. Thus one line is added to the DYMCA RUN.INITIALIZE routine (line 51) to add five signals to the total system signal count. Figure 5.3 shows all of the signal associated with the TANK program. The five new signals are indicated by stars. These signals are then initialized by the TANK.INITIALIZE.RUN routine. Once created, the signals are treated in the same manner as all other component signals. The five signals concerned are the three control signals from the tank to each of the three units, the output process flow from the tank to unit one, and a system status signal to indicate system success or failure.

The TANK.INITIALIZE.RUN routine also creates the signal and component files necessary for clean operation of the program code. The TANK.INITIALIZE.TRIAL routine, which is executed prior to each trial, resets the net flow rate to zero, sets the tank fluid level back to zero, turns the flow out of the tank on, resets the system success indicator to "good," and turns off the command signals to all three control units.

The STOP.TANK process operates in much the same fashion as the STOP.SCENARIO process. It is used to suspend operation of the tank, if the tank has not failed during the simulated time period (which has a very low probability of occurrence), and then to reset the tank so it is ready to be started at the beginning of the next Monte Carlo trial.

Minor modifications were also made to the MAIN routine and the CALL.UPDATE process of the DYMCA program. The MAIN routine was modified to include calling the tank initialization routines and to call the STOP.TANK

process. In addition the availability data structure was modified to print out the desired results in the output file. The CALL.UPDATE process was revised to include lines 14 and 15 which simply take the tank out of its suspended state and cause it to start operation at the beginning of every trial.

In addition, many lines were added to the PREAMBLE to reflect all of the new routines, processes, and variables associated with the TANK program. These lines are indicated in the Preamble listing for the DYMCAm program in Appendix B by the marker "TANK" which is placed at the far right hand side of each line of code which was modified or added. The entire TANK program, as a unit, was compiled and kept separate from the DYMCAm program, since subroutines cannot be compiled separately, and the two codes are not used together. They do, however, contain the same basic structure and the TANK program should be viewed as an extension of the DYMCAm program, which remains almost entirely intact in the TANK code.

The input file necessary to run the program is exactly the same format as the input file for the DYMCAm program described in Appendix A. The only point to note is that the three units were modeled as valves in the simulation program. It is also important that the names of the level control units be entered as unit1, unit2, and unit3 so that they are recognized by the TANK program as the flow control units. An example input file for this program is contained in Appendix D. The same input file is used for all tests, and changes are made in the program to reflect testing for the failure condition of overflow or dryout and to alter the flow rate provided by unit three. The output file generated by the TANK program is identical in format to the output generated by the DYMCAm program, and an example print out is shown in Appendix E.

5.4 TANK Results

Two example cases were considered in the testing process. In the notation of reference A-1, Case A involves unit three having a flow rate of 0.01 meters per minute and in Case F the flow rate from unit three is changed to 0.005 meters per minute. Otherwise the test cases are exactly the same. The change made to reflect the different flow rate is made in the FLOW.UPDATE routine. To test for the probability of tank overflow, lines 13 through 17 of the TANK.UPDATE routine were rendered un-executable, and when testing for tank dryout, lines 13 to 17 were restored and lines 24 through 28 of the Tank.Update routine were removed. This method was used to test for the selected type of failure event, since both tank overflow and dryout are failures and should not be counted together as failures during the same test run.

As explained earlier, if failures can be assumed to be separated by at least 3.5 hours in Case A and 7 hours in Case F, then it is possible to use a Markov chain to approximate a solution to the problem. This approach involves understanding the feasible failure sequences which can occur in each case. An understanding of the failure sequences also provides insight into the problem solution by simulation methods, so they will be described in detail.

5.4.1 Analysis of Case A

For Case A the tank starts at time zero with all units operational, and units one and two turned on, and unit three turned off. The tank will continue in this state with no change in the tank level until a failure of a control unit occurs. The sequencing of failure is very important so each unit failing first will be considered separately. Figure 5.4 shows the state transition diagram for this system. All states are defined in Table 5.3.

Tank Case A

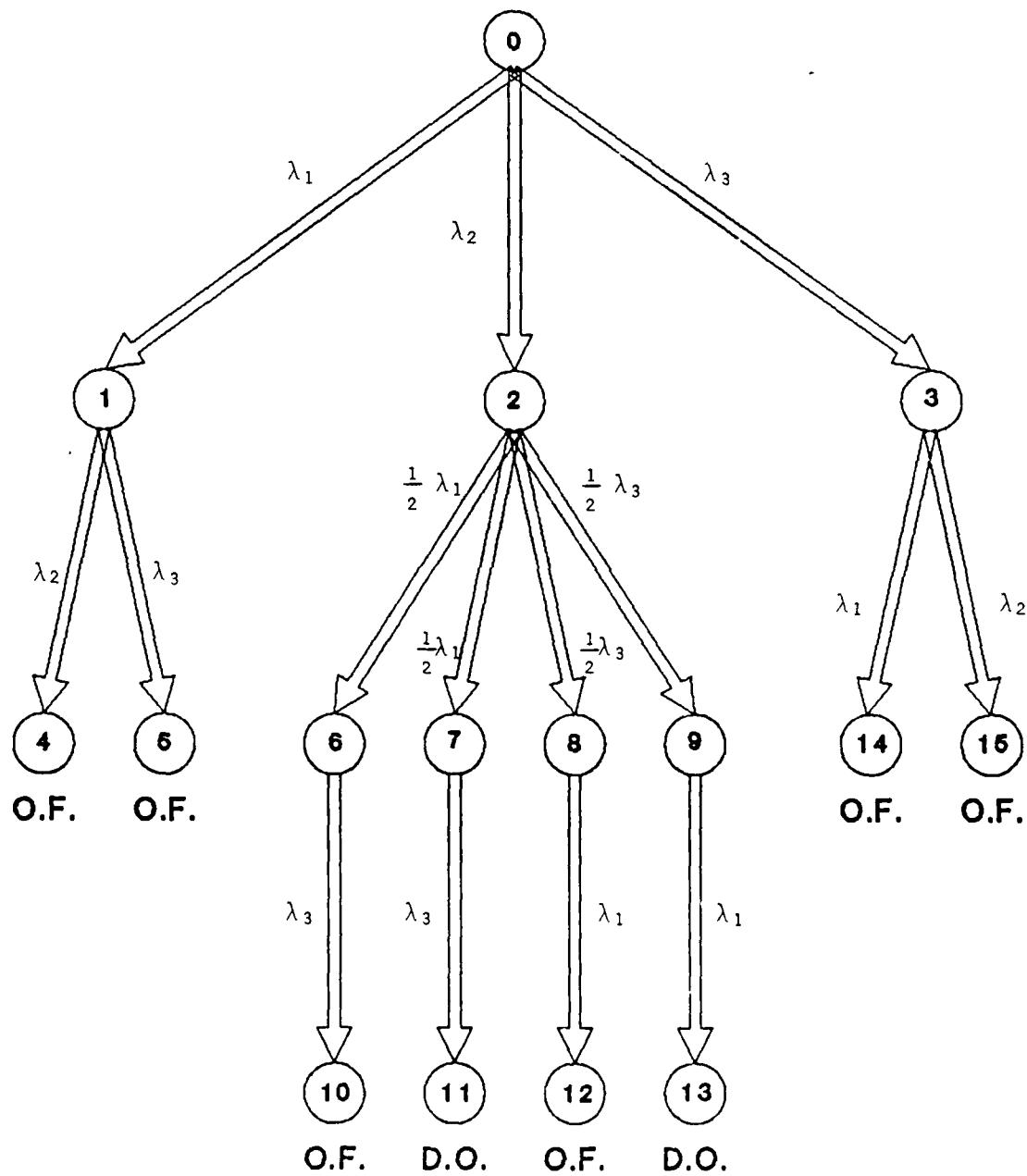


Figure 5.4 Tank Case A State Transition Diagram

Thus the three possible initiating events are unit one or unit two failing closed, or unit three failing open as shown in Figure 5.4. Since the unit which fails first is only dependent on the failure rates of the three units, it seems intuitively clear that the probability of each individual unit being the first to fail is given simply by the ratio of the failure rate for that unit divided by the sum of the failure rates for all three units.

To show this more formally, consider the system composed of only the first four states of Figure 5.2, states 0, 1, 2, and 3. The four Markov equations for this system are:

$$\frac{dP_0}{dt} = -[\lambda_1 + \lambda_2 + \lambda_3] P_0 \quad (5.1)$$

$$\frac{dP_1}{dt} = \lambda_1 P_0 \quad (5.2)$$

$$\frac{dP_2}{dt} = \lambda_2 P_0 \quad (5.3)$$

$$\frac{dP_3}{dt} = \lambda_3 P_0 \quad (5.4)$$

Noting that at time $t=0.0$ the system is initially in state 0 giving $P_0(0)=1.0$, equation 5.1 can be solved to give:

$$P_0(t) = \exp[-(\lambda_1 + \lambda_2 + \lambda_3)t] \quad (5.5)$$

Substituting this result into equation 5.2 gives:

$$\frac{dP_1}{dt} = \lambda_1 \exp[-(\lambda_1 + \lambda_2 + \lambda_3)t] \quad (5.6)$$

which can be solved to yield:

$$P_1(t) = \left[\frac{-\lambda_1}{[\lambda_1 + \lambda_2 + \lambda_3]} \right] \exp [-[\lambda_1 + \lambda_2 + \lambda_3]t] + C, \quad (5.7)$$

where $C = \frac{\lambda_1}{[\lambda_1 + \lambda_2 + \lambda_3]}$, since $P_1(0) = 0.0$.

The equation for state 1 can therefore be written:

$$P_1(t) = \left[\frac{\lambda_1}{[\lambda_1 + \lambda_2 + \lambda_3]} \right] \{1 - \exp [-[\lambda_1 + \lambda_2 + \lambda_3]t]\} \quad (5.8)$$

Using the same solution approach for states 2 and 3 it is found:

$$P_2(t) = \left[\frac{\lambda_2}{[\lambda_1 + \lambda_2 + \lambda_3]} \right] \{1 - \exp -[(\lambda_1 + \lambda_2 + \lambda_3)t]\} \quad (5.9)$$

$$P_3(t) = \left[\frac{\lambda_3}{[\lambda_1 + \lambda_2 + \lambda_3]} \right] \{1 - \exp -[(\lambda_1 + \lambda_2 + \lambda_3)t]\} \quad (5.10)$$

Solving equations 5.8, 5.9, and 5.10 for t sufficiently large, it is clear that:

$$P_1 = \frac{\lambda_1}{[\lambda_1 + \lambda_2 + \lambda_3]} \quad (5.11)$$

$$P_2 = \frac{\lambda_2}{[\lambda_1 + \lambda_2 + \lambda_3]} \quad (5.12)$$

$$P_3 = \frac{\lambda_3}{[\lambda_1 + \lambda_2 + \lambda_3]} \quad (5.13)$$

Using these results it is found that unit three will fail first 43% of the time, unit two 34%, and unit one 23% of the time.

The initial failure of unit one is the easiest case to consider since it will always lead eventually to a tank overflow condition, regardless of the relative flow rates provided by the three units. Unit one failing closed causes the fluid level to rise until it passes into control region number three, at which time unit two is shut off. The tank remains in this

condition until either unit two or unit three fails open, either of which will lead directly to a tank overflow condition.

The initial failure of unit two poses a more interesting problem. With unit two failing closed, the fluid level will drop until it reaches control region number one. Then unit one is closed and unit three is opened. This causes the fluid level to rise until the fluid level is in control region two again, at which time unit one is opened and unit three is closed. Thus the fluid level will continue to oscillate about the low level set point of -1 meters with units one and three being alternately turned on and off. The continuous routine in SIMSCRIPT uses a finite but variable time step, the minimum and maximum of which must be specified by the programmer. For both cases considered, the minimum and maximum time steps were both set equal to approximately one hour, therefore for this case the level of the tank will fluctuate between -0.4 meters and -1.6 meters, spending equal time in each of the two control regions (one and two). This is true since while the level is rising, the rate of increase is 0.01 meters per minute, and while the level is falling the rate of level change is also 0.01 meters per minute. Fluctuation occurs between the same two points since time steps were forced to be constant at one hour intervals.

From this state there are four possible events which can occur. While the fluid level is rising, unit one can fail open or unit three can fail closed, or while the fluid level is decreasing unit one can fail closed or unit three can fail open. It is clear that if either unit fails while the level is rising the flow rates in and out of the tank will then be equal and the fluid level will stop changing until the failure of the third level control unit. This third failure will lead directly to the tank running dry.

If one of the two control units fails while the tank level is dropping

then, again, the tank fluid level will cease to change until the failure of the third unit. This time, the third unit failing will lead directly to overflow of the tank. Since the tank spends an equal time in the rising and falling level states, it is equally likely that the tank will fail in an overflow or dryout state. Thus for the case of unit two being the initial failure event, there is a 50% probability that the tank will fail in each of its two failure conditions.

For the case of unit three failing first, the solution is as easy as for unit one failing first. When unit three fails open the fluid level will begin to rise until the level has reached control region number three at which time unit two will be closed. Now with both units one and three open the fluid level will hold constant at 1 meter. The next failure event, either unit one failing closed or unit two failing open, will lead directly to a tank overflow condition. Thus for all scenarios where unit three fails first, the tank will fail by overflow.

From the above discussion it is evident that all unit one initial failures, all unit three initial failures, and half of the unit two initial failures will eventually lead to an overflow condition. Thus using the values quoted above for the probability that each of the three units will fail first, it is found that the probability that the tank will fail by overflow is:

$$0.23 + 0.43 + (0.5 * 0.34) = 0.83$$

The tank will fail by overflowing approximately 83% of the time and fail by running dry the other 17% of the time.

It is important to note that although the above method simplifies the problem so that it may be solved with Markov chains without even considering the continuously variable tank fluid level, this method is only an

approximation and is as good as the assumption that two failures do not occur within a 3.5 hour time period. This, of course, will not be the case for all continuous variable process control problems. In this example problem the results obtained using the approximation agree well with the simulation results, but several possible failure sequences which will occur with low probability are ignored. For example, consider the case of failure of both units two and three within 1.5 hours of each other. This will leave the fluid level essentially unchanged or, at least, still in control region number two. The net flow rate from the tank is still zero so the tank will remain in this condition until unit one fails, at which time the tank will overflow. If it is considered that unit three fails just prior to unit two, then the result is consistent with the approximate analysis. However if unit two failed first, then the approximate method predicts that half the cases will experience system failure by overflow and half will be by dryout. This is obviously not the case for the dual failure example and the approximate solution will be slightly in error. Other "simultaneous" failures lead to similar conclusions.

5.4.2 Analysis of Case F

For Case F the problem becomes much more complicated. The initial failure probabilities remain unchanged from Case A, but some of the sequences of events after initial failure change. One part that remains the same, however, is the scenario following initial failure of unit number one. Since unit one is the only way fluid can be removed from the tank, once it has failed closed the tank is guaranteed to fail by overflow. Thus as in Case A, if unit one fails first, all scenarios lead to overflow. The time to overflow, however, could be different due to the different flow rate from unit three.

If unit two fails first, the tank level drops to the low set point and begins to oscillate above and below this mark as units one and three are opened and closed (as in Case A). However, the amount of time spent in each control region will be different. When the fluid level is rising, unit one is closed and unit three is open, thus the level is changing at the rate of 0.005 meters per minute. When the level is falling, unit one is open and unit three is closed, thus the level is changing at 0.01 meters per minute. Define the flow rates from each of the three units as \dot{x}_1 , \dot{x}_2 , and \dot{x}_3 respectively. For Case F the normal values are, $\dot{x}_1=0.01$, $\dot{x}_2=0.01$, and $\dot{x}_3=0.005$ meters per minute. Since unit two has failed closed, then $\dot{x}_2=0.0$. Define the net flow rate as \dot{x}_{net} , then while the water level is in control region one (and unit one is closed), \dot{x}_{net} is given by:

$$\dot{x}_{\text{net}} = \dot{x}_3 - 0.005$$

While the water level is in control region two (and unit three is closed), \dot{x}_{net} is given by:

$$\dot{x}_{\text{net}} = -\dot{x}_1 = -0.01$$

Therefore, if the tank level is considered to vary between the same two levels, the tank must spend twice as much time in the control region one (with unit three open and one closed), than in the control region two (with one open and three closed). This will reflect in the failure scenarios.

If while the tank level is increasing, either unit fails, then the tank will immediately run dry. This is the same result as for Case A except that Case A would not experience dryout until all three units have failed. If while the tank level is decreasing, unit one fails, then the tank level will hold constant until unit three fails open. Then the tank will overflow. This sequence is the same as for Case A, however overflow will occur a few hours later due to the slower flow rate from unit three.

The fourth possible failure sequence resulting from the initial failure of unit two is entirely different. If unit three fails while the tank level is decreasing, then the level will continue to decrease until the level reaches control region one, since the flow through unit three is half the value of the flow through unit one. Once in control region one, unit one is closed and the level will rise because of the flow from failed unit three. Once the level is again in control region two, unit one will be opened. Thus the level oscillates about the -1 meter level with equal time spent while the tank level is rising and falling due to the fact that the flow rate from unit one is exactly twice that from unit three. Since unit two has failed closed and unit three has failed open, $\dot{x}_2 = 0.0$ and $\dot{x}_3 = 0.005$. While the water level is in control region one, unit one is closed; x_{net} is given by:

$$\dot{x}_{net} = \dot{x}_3 = 0.005$$

While the water level is in control region two, unit one is open thus x_{net} is given by:

$$\dot{x}_{net} = \dot{x}_3 - \dot{x}_1 = 0.005 - 0.01 = -0.005$$

The water level rises and falls at equal rates.

From this condition, unit one can either fail open or closed depending on whether it fails while the tank level is rising or falling. These failures occur with equal probability. Therefore, once units two and three have failed there is an equal chance that the tank will run dry or overflow.

Summarizing the possible sequences following failure of unit two it is seen that the probability of subsequent failure of unit one or three is equal to the ratio of their failure rates to the sum of the failure rates. Thus there is a 65% chance that the next failure will be of unit three and a 35% chance that the next failure will be of unit one. Of these percentages, two

thirds of the unit one failures will be unit one failing open, which leads directly to dryout, and the other one third of the unit one failures lead to eventual tank overflow. For the unit three failure cases, two thirds will be unit three failing closed, while the fluid level is rising, and this leads to the tank failing by dryout. The other one third lead to oscillation in the fluid level with unit one opening and closing, thus 50% will lead to eventual system overflow and 50% will lead to system dryout. Evaluating the probabilities of the scenarios initiated by the failure of unit two, it is found that 77% lead to tank dryout while 23% lead to tank overflow. Figure 5.5 shows the state transition diagram for the Case F tank problem.

In Case F it is also no longer true that the initial failure of unit three will eventually lead to tank overflow. To see this, the scenarios associated with the initial failure of unit three are analyzed. Following failure of unit three the tank level rises into control region three and then unit two is closed. Since the flow rate from unit one is greater than the flow rate of unit three, the level drops into control region two, at which point unit two is turned back on. Thus the fluid level oscillates about the +1 meter level with unit two being opened and closed. While unit two is on, the net flow rate into the tank is 0.005 meters per minute, and while unit two is off the flow rate out of the tank is 0.005 meters per minute, thus if the tank level is assumed to oscillate between the same two levels, the system spends equal time with unit two open or closed.

The next failure of either unit one or two will again be in proportion to the failure rates associated with each unit. Using these values it is found that subsequent to failure of unit three, there is a 41% chance that the next failure will be of unit one and a 59% chance that the next failure will be of unit two. If unit one fails, it closes, thus the tank will go

immediately to the overflow condition. Since unit two spends fifty percent of its time open and fifty percent of its time closed, it has an equal probability of failing either closed or open.

If while the tank level is decreasing, unit two fails on, the tank will go directly to an overflow state. If, however, unit two fails closed while the tank level is increasing, then the tank level will fall until it is in control region one, at which time unit one will be closed. Then the level will rise due to flow from unit three until the level is in control region two, when unit one will be opened again. Thus the level oscillates about the -1 meter level with unit one opening and closing.

The flow rate when fluid is leaving the tank is the same as the flow rate when the fluid level is rising, therefore unit one spends an equal amount of time open and closed. If unit one fails closed while it is open, then the tank will overflow. If unit one fails open while it is closed, then the tank will run dry. The latter case was not possible in Case A.

Table 5.3

Case A Failure Sequence Summary

| Failure Sequence | Probability | Result |
|-------------------------------|-------------|----------|
| #1 closed, #2 open | 0.10 | overflow |
| #1 closed, #3 open | 0.13 | overflow |
| #2 closed, #1 closed, #3 open | 0.06 | overflow |
| #2 closed, #1 open, #3 closed | 0.06 | dryout |
| #2 closed, #3 open, #1 closed | 0.11 | overflow |
| #2 closed, #3 closed, #1 open | 0.11 | dryout |
| #3 open, #1 closed | 0.17 | overflow |
| #3 open, #2 open | 0.25 | overflow |

Summarizing the scenarios following the initial failure of unit three it is seen that all but one of the situations leads to a tank overflow condition. If unit one fails second, then overflow is certain to occur while if unit two fails second only three quarters of the time will overflow occur.

Evaluating numerically, following the initial failure of unit number three, there is a 85% chance that the tank will fail by overflow and only a 15% chance that the tank will run dry.

Table 5.4

Case F Failure Sequence Summary

| Failure Sequence | Probability | Result |
|-------------------------------|-------------|----------|
| #1 closed, #2 open | 0.10 | overflow |
| #1 closed, #3 open | 0.13 | overflow |
| #2 closed, #1 open | 0.08 | dryout |
| #2 closed, #1 closed, #3 open | 0.04 | overflow |
| #2 closed, #3 open, #1 closed | 0.04 | overflow |
| #2 closed, #3 open, #1 open | 0.04 | dryout |
| #2 closed, #3 closed, #1 open | 0.15 | dryout |
| #3 open, #1 closed | 0.17 | overflow |
| #3 open, #2 open | 0.13 | overflow |
| #3 open, #2 closed, #1 open | 0.06 | dryout |
| #3 open, #2 closed, #1 closed | 0.06 | overflow |

Compiling the results of all initial failure events and evaluating the numerical results it is found that for Case F, the probability that the tank will fail by running dry is 0.30 and the probability that the tank will fail by overflowing is 0.70. Thus in Case F the tank is more likely to fail by running dry than in Case A due to the decreased flow rate from unit three. Table 5.3 summarizes the possible failure sequences for Case A, their probability of occurrence, and the end result. Table 5.4 summarizes the same results for Case F.

5.4.3 Simulation Analysis

The results obtained for the asymptotic failure probabilities agree well with the simulation results, as shall be seen, and Case A coincides with the results presented by Aldemir in reference A-1. For Case F, however, results from the simulation method agree with results from the simplified Markov model, but not as closely with Aldemir's predictions. This is due to the fact that the problem treated in this work considers the failure of

components and Aldemir considers failure of the control system for the components. Thus there will be different failure possibilities between the two approaches.

From the above initiating event analysis, and making the assumption that the time required for the fluid level to transit between control regions is negligible, it is possible to construct Markov chains to approximate the time dependent behavior of the system. Figure 5.4 shows the Markov state transition diagram for Case A and indicates that sixteen states are required. Figure 5.5 shows the state transition diagram for Case F, which requires nineteen states. Table 5.5 shows the states used for Case A and their corresponding definition. States 11 and 13 correspond to tank dryout while states 4, 5, 10, 12, 14, and 15 correspond to tank overflow. From the states in this table the Markov equations are written using the failure rates for each control unit. The Markov equations are shown in Table 5.6.

Table 5.5

Markov States for Tank Case A

| STATE | FAILURE DESCRIPTION |
|-------|--|
| 0 | All units good |
| 1 | Unit 1 failed closed |
| 2 | Unit 2 failed closed |
| 3 | Unit 3 failed open |
| 4 | Unit 1 failed closed then Unit 2 failed open (Overflow) |
| 5 | Unit 1 failed closed then Unit 3 failed open (Overflow) |
| 6 | Unit 2 failed closed then Unit 1 failed closed |
| 7 | Unit 2 failed closed then Unit 1 failed open |
| 8 | Unit 2 failed closed then Unit 3 failed open |
| 9 | Unit 2 failed closed then Unit 3 failed closed |
| 10 | Unit 2 failed closed then Unit 1 failed closed then Unit 3 failed open (Overflow) |
| 11 | Unit 2 failed closed then Unit 1 failed open then Unit 3 failed closed (Dryout) |
| 12 | Unit 2 failed closed then Unit 3 failed open then Unit 1 failed closed (Overflow) |
| 13 | Unit 2 failed closed then Unit 3 failed closed then Unit 1 failed open (Dryout) |
| 14 | Unit 3 failed open then Unit 1 failed closed (Overflow) |
| 15 | Unit 3 failed open then Unit 2 failed open (Overflow) |

Tank Case F

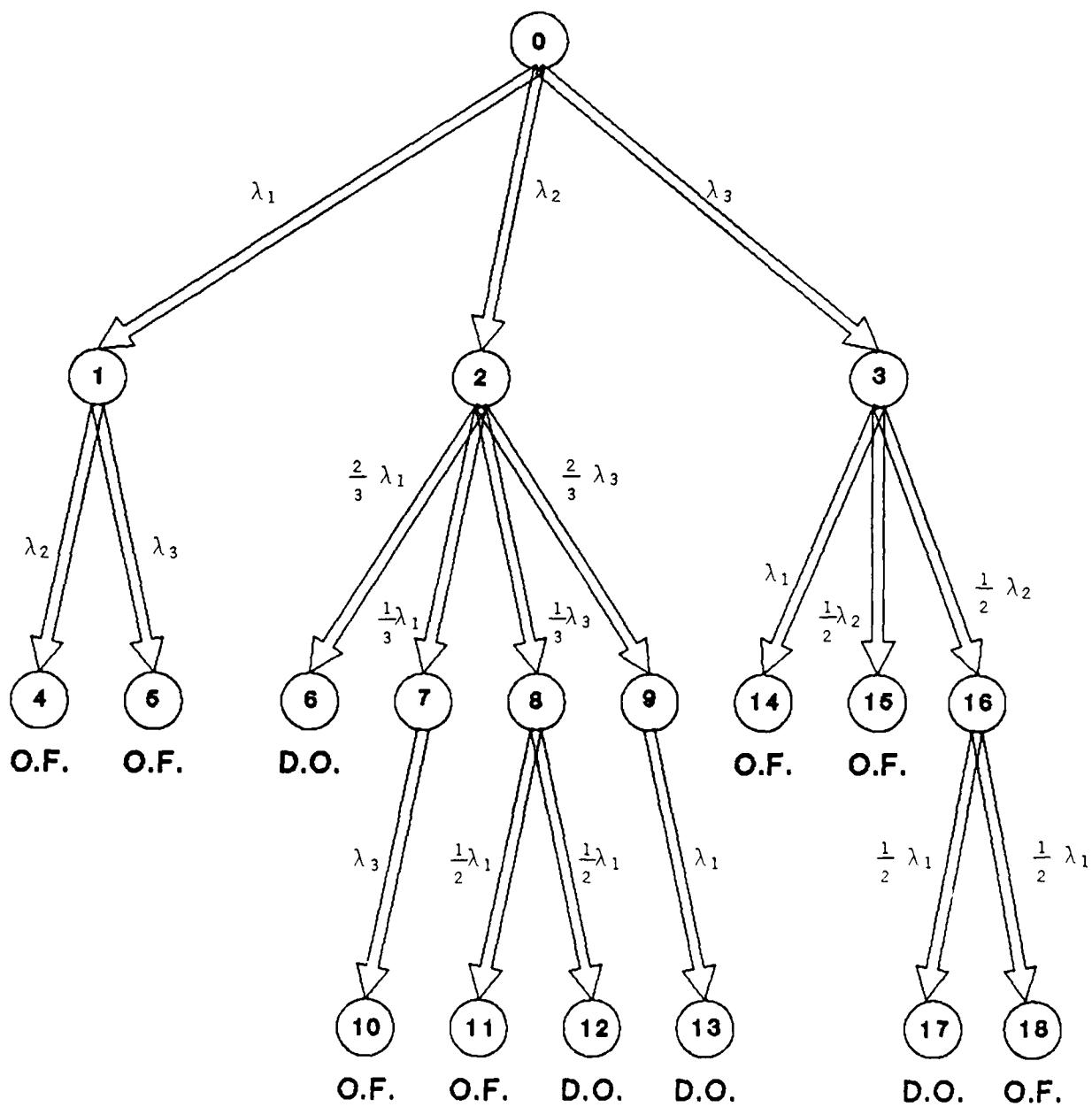


Figure 5.5 Tank Case F State Transition Diagram

Table 5.6
Markov Equations for Tank Case A

$$\begin{aligned}
 \frac{dP_0}{dt} &= - [\lambda_1 + \lambda_2 + \lambda_3] P_0 \\
 \frac{dP_1}{dt} &= - [\lambda_2 + \lambda_3] P_1 + \lambda_1 P_0 \\
 \frac{dP_2}{dt} &= - [\lambda_1 + \lambda_3] P_2 + \lambda_1 P_0 \\
 \frac{dP_3}{dt} &= - [\lambda_1 + \lambda_2] P_2 + \lambda_1 P_0 \\
 \frac{dP_4}{dt} &= \lambda_2 P_1 \\
 \frac{dP_5}{dt} &= \lambda_3 P_1 \\
 \frac{dP_6}{dt} &= - \lambda_3 P_6 + 0.5 \lambda_1 P_2 \\
 \frac{dP_7}{dt} &= - \lambda_3 P_7 + 0.5 \lambda_1 P_2 \\
 \frac{dP_8}{dt} &= - \lambda_1 P_8 + 0.5 \lambda_3 P_2 \\
 \frac{dP_9}{dt} &= - \lambda_1 P_9 + 0.5 \lambda_3 P_2 \\
 \frac{dP_{10}}{dt} &= \lambda_3 P_6 \\
 \frac{dP_{11}}{dt} &= \lambda_3 P_7 \\
 \frac{dP_{12}}{dt} &= \lambda_1 P_8 \\
 \frac{dP_{13}}{dt} &= \lambda_1 P_9 \\
 \frac{dP_{14}}{dt} &= \lambda_1 P_3 \\
 \frac{dP_{15}}{dt} &= \lambda_2 P_3
 \end{aligned}$$

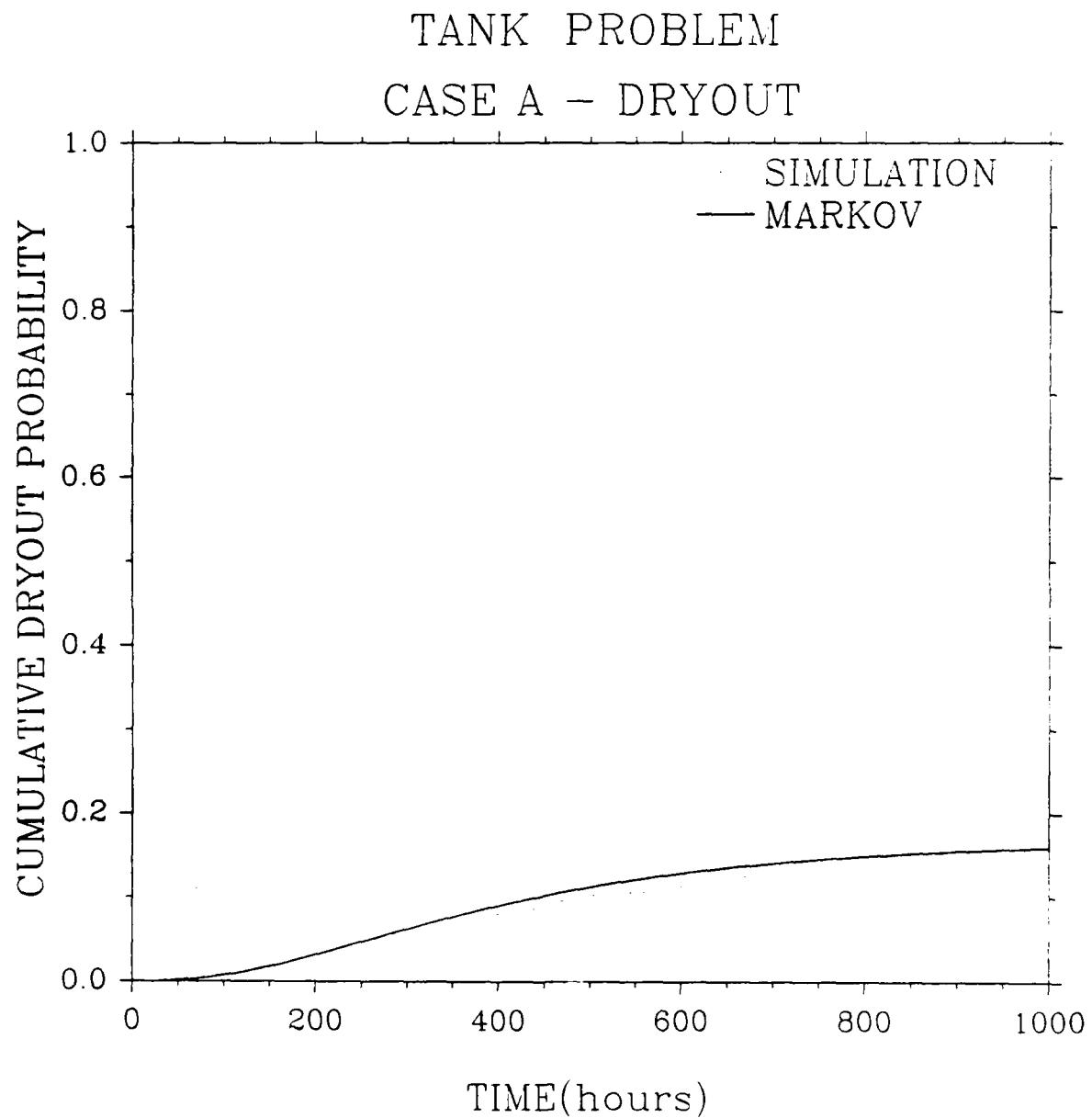


Figure 5.6 Case A - Cumulative Dryout Probability

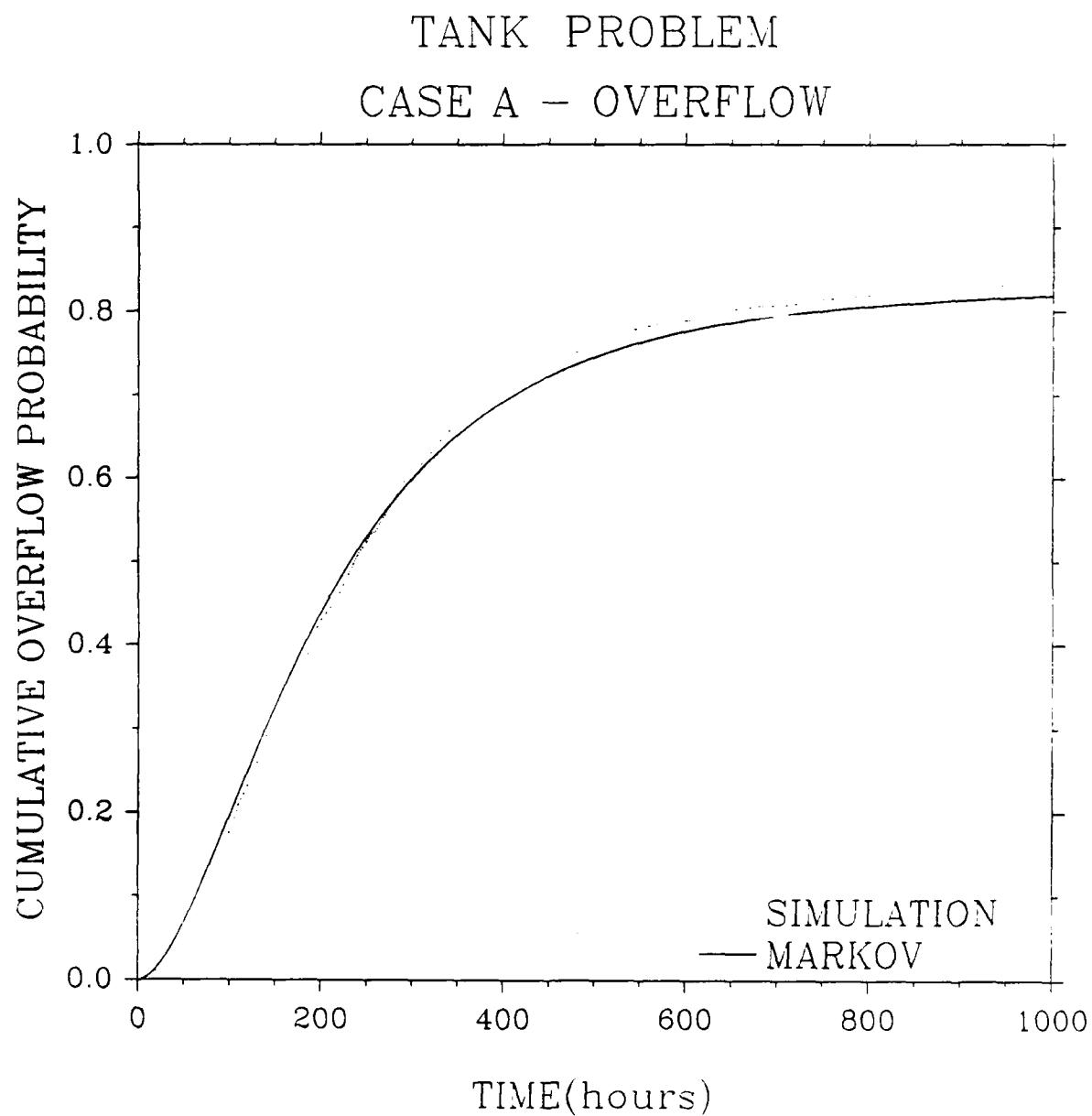


Figure 5.7 Case A - Cumulative Overflow Probability

The failure rates are defined in reference A-1 as:

$$\text{Unit 1: } \lambda_1 = 5.2 * 10^{-5} \text{ per minute}$$

$$\text{Unit 2: } \lambda_2 = 7.6 * 10^{-5} \text{ per minute}$$

$$\text{Unit 3: } \lambda_3 = 9.5 * 10^{-5} \text{ per minute}$$

To solve these equations, a fourth order Runge-Kutta numerical integration routine (obtained from ref. P-4) was used as in Chapter 4. The time period of concern is from time zero up until approximately 1,000 hours. The time dependent results for appropriate states were summed to obtain the time dependent probability of system failure by overflow or by dryout.

The TANK program was run for a simulated time duration of 1,000 hours and 1,000 Monte Carlo trials were performed. The results were then plotted along with the Markov approximation for comparison. Figure 5.6 shows the time dependent results for the tank running dry and Figure 5.7 shows the results for the tank failing by overflow. Both of these figures indicate good agreement between the simulation results and the Markov approximation. The time dependent behavior is virtually identical and values differ by only a few percent. Good agreement between the simulation results and the simplified Markov model is expected since the time required for the tank level to change is small in comparison with the failure times associated with the individual flow control units. A quantitative comparison of the simulation and simplified Markov results with the numerical results provided by Aldemir's dynamic Markov approach for Case A is shown in Figure 5.10. The data for Aldemir's approach was provided in reference A-5, and is the same as the results presented in reference A-1. The figure indicates that the simplified Markov results agree almost exactly with Aldemir's predictions and the simulation method provides results which are very similar to both.

For this case, the difference between the three methods is very small and indicates that although the approach to the problem was different for each method, the results were quite comparable.

For approximate analysis of Case F using the Markov technique, there are nineteen states of interest. These states are listed in Table 5.7 below. States 6, 12, 13, and 17 contribute to tank dryout while states 4, 5, 10, 14, 15, and 18 contribute to tank failure by overflow.

Table 5.7
Markov States for Tank Case F

| STATE | FAILURE DESCRIPTION |
|-------|--|
| 0 | All units good |
| 1 | Unit 1 failed closed |
| 2 | Unit 2 failed closed |
| 3 | Unit 3 failed open |
| 4 | Unit 1 failed closed then Unit 2 failed open (Overflow) |
| 5 | Unit 1 failed closed then Unit 3 failed open (Overflow) |
| 6 | Unit 2 failed closed then Unit 1 failed open (Dryout) |
| 7 | Unit 2 failed closed then Unit 1 failed closed |
| 8 | Unit 2 failed closed then Unit 3 failed open |
| 9 | Unit 2 failed closed then Unit 3 failed closed |
| 10 | Unit 2 failed closed then Unit 1 failed closed then Unit 3 failed open (Overflow) |
| 11 | Unit 2 failed closed then Unit 3 failed open then Unit 1 failed closed (Overflow) |
| 12 | Unit 2 failed closed then Unit 3 failed open then Unit 1 failed open (Dryout) |
| 13 | Unit 2 failed closed then Unit 3 failed closed then Unit 1 failed open (Dryout) |
| 14 | Unit 3 failed open then Unit 1 failed closed (Overflow) |
| 15 | Unit 3 failed open then Unit 2 failed open (Overflow) |
| 16 | Unit 3 failed open then Unit 2 failed closed |
| 17 | Unit 3 failed open then Unit 2 failed closed then Unit 1 failed open (Dryout) |
| 18 | Unit 3 failed open then Unit 2 failed closed then Unit 1 failed closed (Overflow) |

With the above definitions of states and the same definition of failure rates as given for Case A, the Markov equations are obtained and shown in Table 5.8. Figure 5.5 shows the state transition diagram for these equations.

Table 5.8
Markov Equations for Tank Case F

$$\begin{aligned}
 \frac{dP_0}{dt} &= -[\lambda_1 + \lambda_2 + \lambda_3]P_0 \\
 \frac{dP_1}{dt} &= -[\lambda_2 + \lambda_3]P_1 + \lambda_1 P_0 \\
 \frac{dP_2}{dt} &= -[\lambda_1 + \lambda_3]P_2 + \lambda_1 P_0 \\
 \frac{dP_3}{dt} &= -[\lambda_1 + \lambda_2]P_3 + \lambda_1 P_0 \\
 \frac{dP_4}{dt} &= \lambda_2 P_1 \\
 \frac{dP_5}{dt} &= \lambda_3 P_1 \\
 \frac{dP_6}{dt} &= \left[\frac{2}{3}\right] \lambda_1 P_2 \\
 \frac{dP_7}{dt} &= -\lambda_3 P_7 + \left[\frac{1}{3}\right] \lambda_1 P_2 \\
 \frac{dP_8}{dt} &= -\lambda_1 P_8 + \left[\frac{1}{3}\right] \lambda_3 P_2 \\
 \frac{dP_9}{dt} &= -\lambda_1 P_9 + \left[\frac{2}{3}\right] \lambda_3 P_2 \\
 \frac{dP_{10}}{dt} &= \lambda_3 P_7 \\
 \frac{dP_{11}}{dt} &= 0.5 \lambda_1 P_8 \\
 \frac{dP_{12}}{dt} &= 0.5 \lambda_1 P_8 \\
 \frac{dP_{13}}{dt} &= \lambda_1 P_9 \\
 \frac{dP_{14}}{dt} &= \lambda_1 P_3 \\
 \frac{dP_{15}}{dt} &= 0.5 \lambda_2 P_3 \\
 \frac{dP_{16}}{dt} &= -\lambda_1 P_{16} + 0.5 \lambda_2 P_3 \\
 \frac{dP_{17}}{dt} &= 0.5 \lambda_1 P_{16} \\
 \frac{dP_{18}}{dt} &= 0.5 \lambda_1 P_{16}
 \end{aligned}$$

Again the fourth order Runge-Kutta numerical integration program (from ref. P-4) was used to solve the equations and the time dependent results for the appropriate states were added together to produce the time dependent probability of tank failure due to overflow and due to dryout. The FLOW.UPDATE routine of the TANK program was modified to reflect the change in flow rate from unit three and then the program was run again using the same input file as for Case A to generate the simulation results. The input file is contained in Appendix D and an example output file is in Appendix E. The program was run for a simulated period of 1,000 hours and as in Case A, 1,000 trials were used. Results of the simulation program are plotted in Figures 5.8 and 5.9 along with the Markov predictions for comparison. The results indicate reasonable agreement between the two methods.

Also of note concerning the TANK simulation program is that from the test run to determine failures sequences, it was found that 13 of the 1,000 trials involved failure of two units during the same continuous process integration time step. In other words, in the 1,000 trials, 13 times, the time separating successive failures was one hour or less. Thus approximately 1.3 percent of the time the assumption made for the initiating event Markov analysis is not valid.

Also of note for the simulation method is the computer time required to complete the problem. Using an integration time step of one hour, running the problem for a simulated time period of 1,000 hours, and performing 1,000 trials caused, the program to take two hours and fifty minutes for the Case A problem. Using the same parameter with the Case F problem, the test took four hours and forty-three minutes. The time for Case F was much longer because of the fact that in this case there were many more instances where the level of the tank oscillated about either the low or the

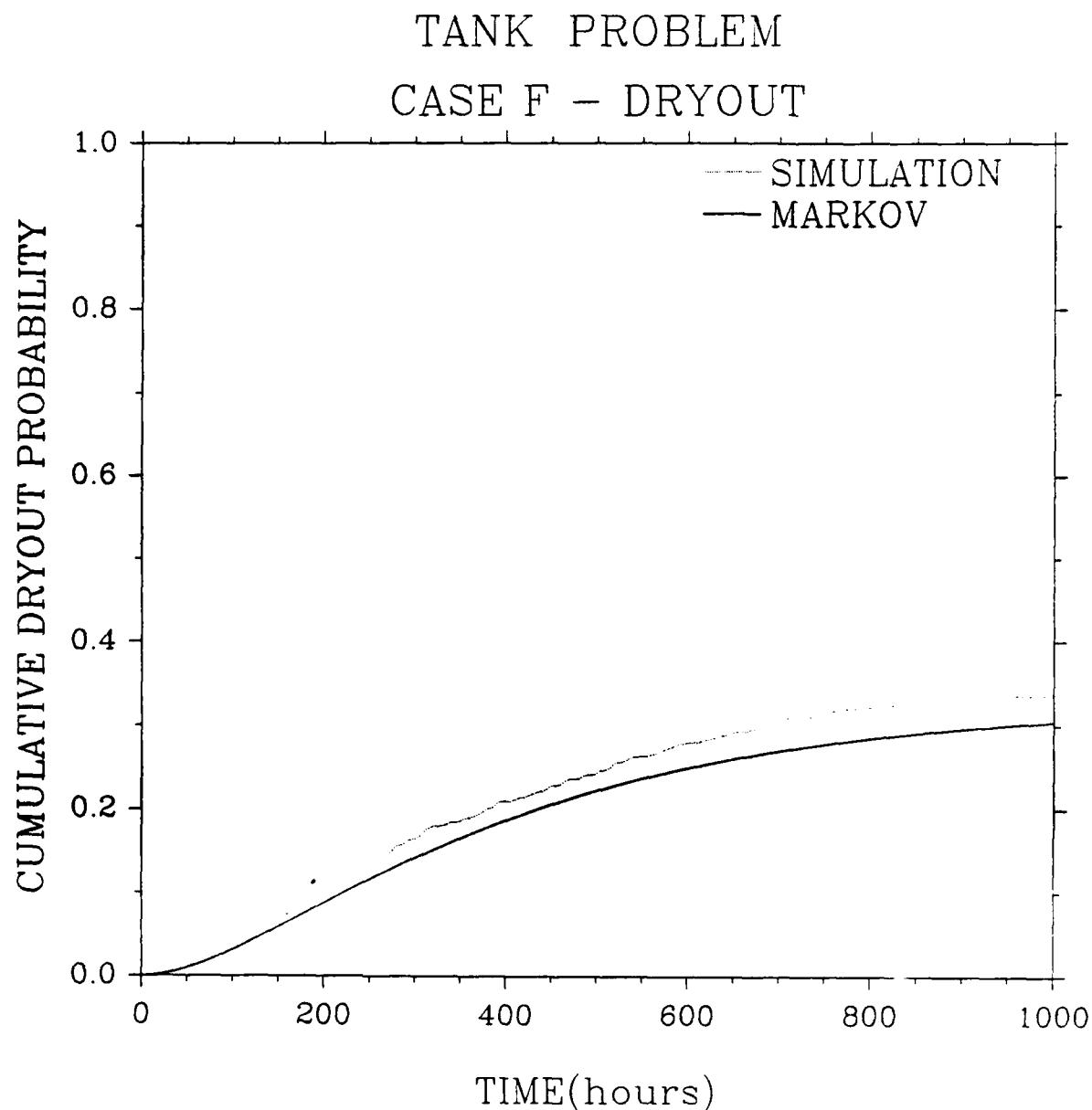


Figure 5.8 Case F - Cumulative Dryout Probability

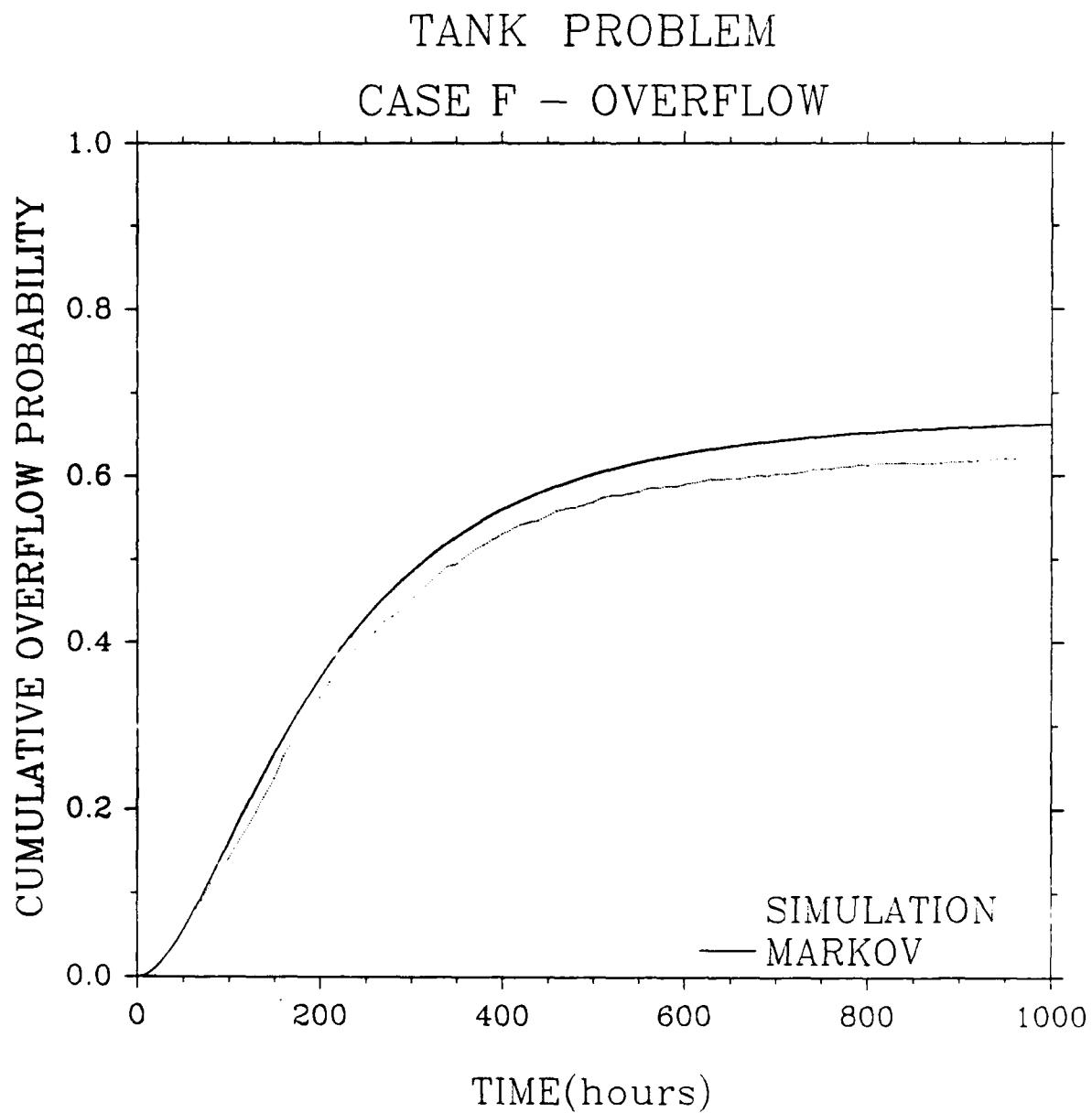


Figure 5.9 Case F - Cumulative Overflow Probability

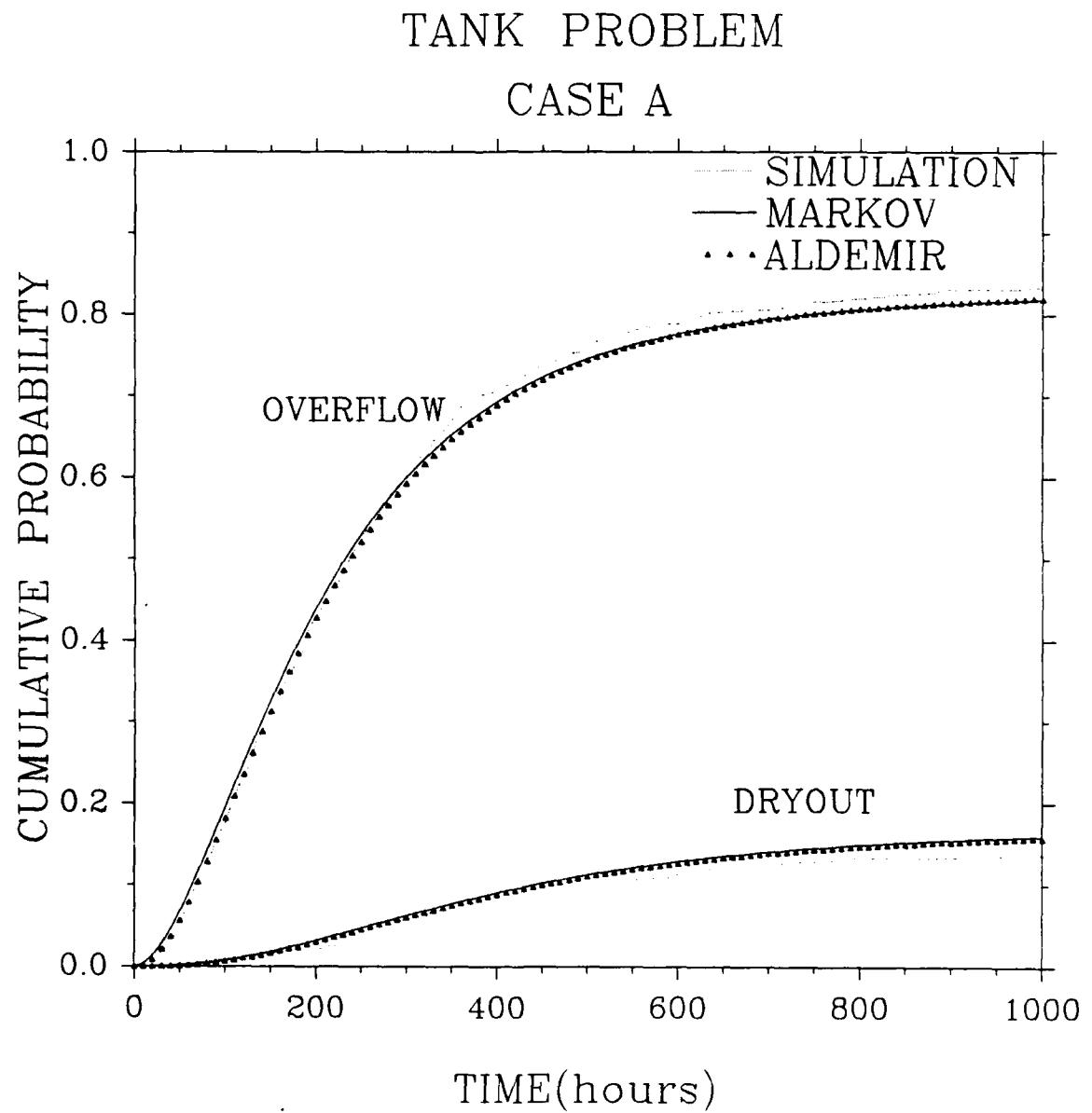


Figure 5.10 Comparison With Aldemir's Results for Case A

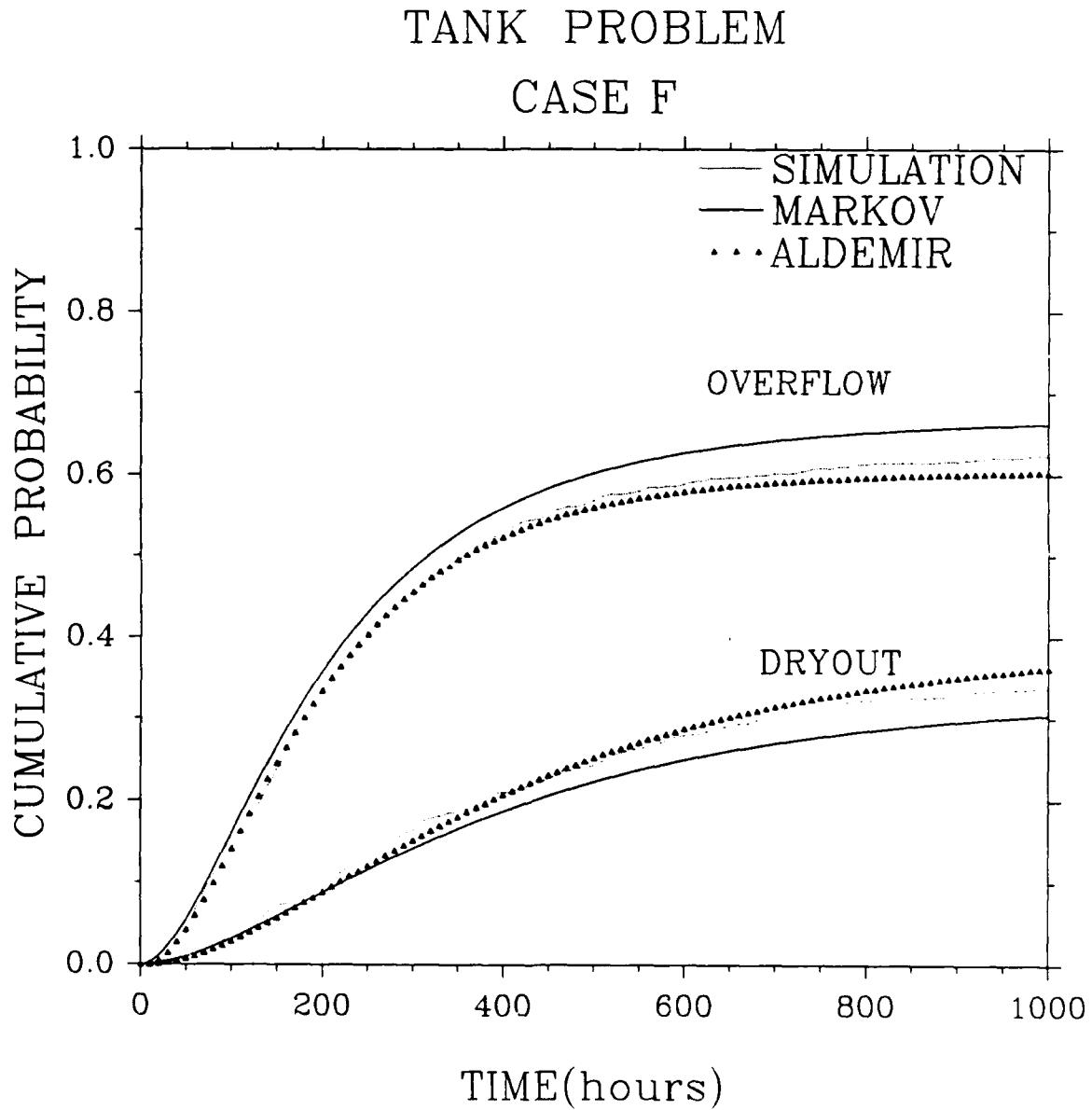


Figure 5.11 Comparison With Aldemir's Results for Case F

high tank level set points. The time stated above is for runs on a COMPAQ 386 personnel computer and times on an IBM XT are estimated to be about six times as long. Thus the time requirement for using this simulation method may be prohibitive.

Although the problem solved in the simulation approach was slightly different than the one solved by Aldemir, results are compared in Figure 5.11. It is seen that the simplified Markov model now has an observable error due to the assumptions made in its development. The simulation results, however still show reasonable agreement with Aldemir's predictions using the dynamic Markov model. Note that the data plotted for Aldemir's model are obtained from reference A-5 and are corrected versions of the data presented in reference A-1.

Certain improvements may be possible to improve the computer time required. One of these, is increasing the time of the integration time step used by the continuous process routine. Another is to more efficiently code the portions of the model which lead to oscillation of a component. Based on the difference in time required for Case A and Case F, this improvement alone may reduce solution time by 75% or more. Other techniques of optimizing the computer code may also certainly be possible as the code was written to be transparent, not necessarily efficient. It is evident that the continuous simulation routine is a valuable tool, however improvements can be made to increase the accuracy of results and to reduce the amount of computer time required.

5.6 Chapter Summary

In this chapter the use of continuous simulation methods was explored for use in analyzing the reliability of complex process control systems. The specific problem investigated was the tank level control problem addressed

by Aldemir in reference A-1. The simulation solution proposed was a modified version of the DYMCA program discussed in previous chapters. This new program, called TANK, made use of the continuous capability available in the SIMSCRIPT II.5 simulation language.

The tank level control problem addressed by Aldemir was discussed in detail to provide insight into the exact nature of a simulation problem. The Case A and Case F scenarios were explored and all possible failure sequences were identified. An assumption was made concerning the time between failure events which allowed for an approximate solution to be developed against which the results of the simulation approach could be compared.

In the third section of the chapter, the modifications made to create the TANK program were described. For the most part, the DYMCA program was left intact with only minor changes being made to a few lines of the SIMSCRIPT code. Several routines were added to define the continuous variable to be used in the simulation. These new routines include a Tank process which models the fluid level as a continuous variable, and monitors the level to determine the control region the system is in, and based on this information, causes the opening and closing of control valves. This type of dynamic problem is not treated by most reliability analysis techniques.

Once the program has been explained, the approximate Markov method is described in detail. The Markov states used for both Case A and Case F are listed in Tables 5.5 and 5.6 respectively and the Markov equations used are listed. These equations were solved using a fourth order Runge-Kutta numerical integration technique and the resulting time dependent system state information manipulated to provide a time dependent estimate of the probability of the tank failing by overflow or by dryout.

The TANK simulation program was run for a simulated time period of

1,000 hours and for 1,000 Monte Carlo trials to provide the simulation estimate to the tank level control problem. These results were plotted with the results from the approximate Markov chain approach for comparison. It was seen that both methods give similar results for the probability of tank failure by overflow and dryout for both Case A and Case F. The results for Cases A and F also compare well with the results given by Aldemir in reference A-5.

The computer time requirements for running the simulation program on a personnel computer were quite large. This is due in large part to the presence of the oscillation of the fluid level about the upper or lower tank level set points. To reduce computer time requirements, it is possible to revise the code to reflect a more efficient program, and the integration time step can be increased. To increase the accuracy of the results, a larger number of trials must be performed. Since the time required is directly related to the number of trials performed, variance reduction techniques will certainly be necessary. The TANK program demonstrated the capability of using a continuous Monte Carlo simulation technique to solve complex process control system reliability analysis problems with satisfactory approximate results.

Chapter 6

Summary and Conclusions

6.1 Discussion of Methods

To evaluate the availability of a system there are two basic types of approaches. These are static and dynamic methods. Under the heading of static methods, the most well known technique is the fault tree. This method has seen extensive use in reliability analysis and is a valuable tool for calculating average system reliability. Another static method is the GO methodology. This is a computer method which uses inductive logic to achieve reliability analysis results. It's major advantage over the more widely used fault tree method is that it models individual system components and therefore provides a model which is easily reviewable and which can be modified easily to analyze slight variations on the original analysis problem. Both of these methods are good for determining average reliability information, but neither can be easily used to solve dynamic analysis problems.

Many methods exist for solving dynamic system availability problems. One of the most commonly used is Markovian analysis. This method can provide exact analytical continuous-time descriptions of systems which can be modeled by a discrete state space. The major drawback of the technique, are the size of the state space when complex systems are to be considered, and that only exponentially distributed failure and repair time distributions can be used.

A second dynamic analysis method is the event tree. This method provides modelling of the sequence of events which can lead to a designated outcome. The method provides an inductive means of calculating the reliability of a system where initiating faults can lead to unfavorable outcomes. The method does not explicitly model repair and failure cycles of

components and it can not be used to evaluate systems which have loops in system operation which may the analysis to return back to a previous node in the event tree a random number of times.

Digraph techniques provide a means of handling systems with continuous process variables. The method is ideally suited to evaluating process control systems in which the state of components may depend on the value of a continuously varying signal. The results of a digraph analysis provide a listing of disturbances which can lead to undesirable performance of the system being analyzed.

The GO-FLOW methodology is similar to GO, but it provides a dynamic capability. Thus this technique model provides an easily constructable reliability analysis model which can be used to evaluate dynamic systems. However, it can only be used to solve for discrete state systems, and is not directly useful in evaluating process control systems or any structure with continuously variable signals and component states.

The most flexible method of availability analysis, is probably also the least used. This is the method of simulation. Monte Carlo simulation techniques provide a powerful alternative to solving complex system reliability analysis problems. In many cases, simulation can be used to solve problems to which there is no analytical solution. The method can be used to evaluate any type of phased mission problem. Since the model is frequently developed to fit only a particular problem or specific type of problem, often fewer assumptions or approximations are necessary and the model can be made to accurately reflect actual system behavior.

Drawbacks of the Monte Carlo simulation method are that it only provides an estimate of the actual system reliability. The level of uncertainty in the prediction will be a result of the number of Monte Carlo

trials performed and the behavior of the random number generator employed. For better accuracy the number of trials can be increased, but this can lead to a large computer time requirement. Typically, an analytical solution method can produce results in a fraction of the time required for solution by simulation techniques, provided an analytic solution to the problem exists. However, significant time might also be involved in determining the form of the analytical solution for the system.

In this work, a new Monte Carlo simulation model was developed for evaluating the dynamic availability of complex systems. The DYMCAm program is designed to be a general analysis tool with applicability to many types of engineering systems. The SIMSCRIPT II.5 language provides the capability for all three major types of simulation approaches including event scheduling, process interaction, and continuous simulation, thus providing flexibility in program development. All three methods are used in the TANK program.

The basic DYMCAm program is designed to provide a model which can analyze the time-dependent availability of dynamic systems, is easy to construct, and can be easily modified to incorporate additional features as needed. The program structure allows for prediction of time-dependent system unavailability information at any number of user specified time points throughout the course of the simulated time period, and it also provides time averaged unavailability information for the entire simulation time. It has the capability to schedule any number of external events thus providing a limitless phased mission capability. Five basic component types are presently modeled, however further components could easily be added if specific problem requirements call for increased modeling capabilities. Much like the GO and GO-FLOW codes in this respect, DYMCAm should be easy to

employ in system evaluation since it is expected that an input file can be written to evaluate a system using the DYMCA code directly from a schematic of the system.

The TANK code is a modified version of DYMCA designed to demonstrate the capability for evaluating systems containing continuous variables. These systems, such as process control systems, can be quite difficult to evaluate using the analytical analysis tools available. The TANK code provides the ability to model a continuously variable tank fluid level and it also demonstrates how a simulation program can be used to model the occurrence of events not scheduled before the start of the simulation. The DYMCA and TANK codes demonstrate that Monte Carlo computer simulation techniques can be employed to solve a wide variety of system availability analysis problems.

6.2 Discussion of Results

The DYMCA program was first tested on two very basic component availability examples to demonstrate that the program does indeed provide meaningful results. The results obtained are accurate and the variance acceptable for the number of trials performed. The two examples consisted of a single component with exponentially distributed repair and failure distributions and a three state component possessing, in addition to these two states, an exponentially distributed repair delay state. In both cases the results agreed well with analytic predictions.

The second case, involving the three state device, demonstrated a minor capability of the rather powerful DYMCA subroutine called the Repair.Supervisor. This subroutine can be used to cause various types of repair delays and even to control which components are repaired and when. Repair resources can even be limited if this is necessary for analysis of certain systems.

The third example demonstrated solution of a simple two-out-of-three system. Success occurs if two of three parallel aligned pumps are operating and flow is being produced at the outlet valve which all three pumps supply pressure too. Results for this example again agreed well with Markov predictions for the system and further demonstrated the capability of the DYMCAIM program to compute the availability of simple systems. The example also identified the desireable capability of having signal process strength incorporated into the model. Although not currently present, such capabilities could easily be added.

The fourth test of the DYMCAIM program demonstrated a simple phased mission problem. The example used was one taken from reference M-2 and has also been solved using the GO-FLOW methodology. Results obtained with the DYMCAIM program indicated the simulation approach gives availability information equivalent to the values estimated by GO-FLOW. A sensitivity analysis was also performed on this problem to verify the hypothesis that the variance of results decreases with the increasing number of trials performed.

The TANK program was designed to demonstrate the continuous capability of the SIMSCRIPT II.5 simulation approach. Continuous variable modeling is an important aspect of simulation a simulation approach, since few analytical methods can treat such systems adequately. Aldemir proposes a discrete state-space continuous time solution method with probabilistic system behavior simulated by Markov chains in reference A-1. Also in this reference is the specific tank example process control problem addressed in this work.

Using the TANK code, simulation solutions for the unavailability of the tank due to overflow and dryout were calculated for the Case A and Case F scenarios of reference A-1. Results compared favorably with Aldemir's solutions, despite the fact that the component states treated in the TANK

model are somewhat different than the states assumed by Aldemir. A simplified Markov chain solution was also proposed in this work for comparison and the Markov results agreed within reasonable accuracy with the simulation results obtained for both Case A and F. For Case A, the simplified Markov solution provided results that agreed almost exactly with Aldemir's solution, while for Case F the simulation results were in closer agreement with Aldemir's solution than was the simplified Markov approach.

The TANK program demonstrated that simulation of complex process control systems may provide a simple method of solution to a problem which is not readily solved by analytic methods. Results appear to be accurate, and the standard deviation of the results are related directly to the number of trials performed.

Another important function demonstrated was the ease with which a simulation approach can change the state of components based on the state of a process variable or other components in the system, at any time point during a simulated run. This is an important function not easily handled by other reliability analysis techniques. By improving the DYMCAm program and properly exploiting this capability it will be possible to analyze many stochastic systems which were previously not easily quantifiable.

6.3 Strengths and Weaknesses

The DYMCAm dynamic simulation model demonstrated the capability of simulation programs to solve dynamic reliability analysis problems. Values of unavailability can be calculated for a system at any time point during the simulation which the user chooses. In this respect, the program is equal in capability to continuous Markov analysis procedures. Although failure times are treated as exponentially distributed and repair times are Weibull distributed in the DYMCAm program, it is a simple matter to change the

program to use any type of transition distributions.

DYMCAM can also be used to solve any manner of deterministic phased mission problem. Through use of external events, components and signals may be changed at will during the execution of the simulation. Although not incorporated into the basic DYMCAM code, the TANK code example demonstrated that it is even possible to simulate stochastic systems in which components are required to change operating state at time points determined by system operating characteristics. The TANK code also shows that Monte Carlo simulation can be successfully used to solve continuous variable reliability analysis problems such as process control systems.

The major drawback of these simulation techniques are that they are only estimation tools and do not provide exact results as do analytical methods. The accuracy of the estimate improves with the number of Monte Carlo trials performed, however the number of trials necessary to significantly reduce the variance of the estimate may be prohibitively large, requiring unacceptable amounts of computer time. As computers become faster, this may prove to be less of a problem, in which case, in theory, exact results can be obtained by using infinitely many trials, provided the simulation model of the system is an accurate one.

It should be noted that the computer run times discussed in conjunction with the tests of this work should not be interpreted as meaning that simulation methods must always require excessive amounts of time. First, neither DYMCAM nor TANK were programmed for maximum efficiency, but rather to be as transparent as possible to the user. In addition, conversations with individuals from CACI indicate that the IBM/PC version of SIMSCRIPT II.5 does run very slowly. This is because the language was originally developed for mainframe computers, and the PC adaptation uses an interpreter, rather

than a compiler. SIMSCRIPT II.5 will run much faster on a mini-computer.

6.4 Conclusion and Recommendations

It has been shown that Monte Carlo simulation methods provide a powerful tool for solving many types of complex system availability analysis problems. This work introduces a program which can be used to solve a wide variety of problems simply by entering an input file which accurately describes the relationships between components in the system. Many large complex systems have no adequate solution techniques, therefore advances in simulation technology is essential for solving many reliability analysis problems.

As is evident from the variance of the results and computer time required to obtain them, many improvements in the method can be made. Cleaner coding of the program may improve run time requirements to some extent; however a more important area of concern should be in exploring methods of variance reduction. Incorporating such techniques may significantly reduce the need to use many Monte Carlo trials and can, therefore, reduce computer time requirements. Once run time has been significantly reduced, more extensive testing of the program should be done to better determine the limits of the simulation technique.

The program should also be modified to consider the strength of process signals. Currently, signals are either on or off indicating only the presence of a process, not the actual strength. If signal strength capabilities were present then it would be an easy matter to determine, for instance, how many pumps were feeding water to a valve simply by the strength of the process signal from the valve.

Another area for future work is on the Repair Supervisor routine. This process could be expanded to provide limitless capabilities in managing

repair resources available to a system. This routine could be used to control the order and scheduled times of repair for individual components based on any desireable scheduling scheme.

The DYMCA M dynamic simulation model demonstrates the basic capability of Monte Carlo techniques to solve any manner of complex system reliability analysis problems. In the future, as analysis of advanced engineering systems is required, development and application of approaches such as this will become desireable and even necessary since analytic techniques may not be practical or possible. Future improvements of the DYMCA M program should make it a valuable tool for computing availability of dynamic systems.

References

- A-1 T. Aldemir, "Computer-Assisted Markov Failure Modeling of Process Control Systems," IEEE Transactions on Reliability, Vol. R-36, No. 1, (April 1987).
- A-2 T. Aldemir, "Quantifying Setpoint Drift Effects in the Failure Analysis of Process Control Systems," Reliability Engineering & System Safety, Vol. 24, No. 1, (1989).
- A-3 R. N. Allan, Y. A. Jebril, A. Saboury, and J. Roman, "Monte Carlo Simulation Applied to Power System Reliability Evaluation," in System Simulation (10th Advances in Reliability Technology Symposium) Elsevier Applied Science Publishing Company, Inc., New York, 1988.
- A-4 G. E. Apostolakis, S. L. Salem, and J. S. Wu, "CAT: A Computer Code for the Automated Construction of Fault Trees," NP-705 Electric Power Research Institute, (1978).
- A-5 T. Aldemir, personal communication to N. Siu, Massachusetts Institute of Technology, April 10, 1989.
- B-1 R. Billinton and M. Patwardhan, "A Modified GO Methodology for System Availability Assessment," in Poster Session (10th Advances in Reliability Technology Symposium) Elsevier Applied Science Publishing Company, Inc., New York, 1988.
- B-2 R. Billinton and R. N. Allan, Reliability Evaluation of Engineering Systems: Concepts and Techniques, Plenum Press, New York, 1983.
- B-3 A. Bendell, "New Methods in Reliability Analysis," in Reliability Technology: Theory and Applications (European Reliability Conference 1986: Copenhagen, Denmark) Elsevier Science Publishing Company, Inc., New York, 1986.
- B-4 J. Banks and J. S. Carson II, Discrete-Event System Simulation, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.
- B-5 J. Banks and J. S. Carson II, "Process-interaction Simulation Languages," Simulation, Vol. 44, No. 5, (May 1985).
- C-1 CACI, SIMSCRIPT II.5 Programming Language, CACI, Inc.-Federal, Los Angeles, 1987.
- C-2 CACI, PC SIMSCRIPT II.5 Introduction and User's Manual, Third Edition, CACI, Inc.-Federal, Los Angeles, 1987.
- D-1 B. S. Dhillon, Systems Reliability, Maintainability and Management, Petrocelli Books, Inc., New York, 1983.
- D-2 B. S. Dhillon and S. N. Rayapati, "A Complex System Reliability Evaluation Method," Reliability Engineering, Vol 16, No. 2, (1986).

- F-1 A. M. Fayek, Introduction to Combined Discrete-Continuous Simulation Using PC SIMSCRIPT II.5, CACI, Inc.-Federal, Los Angeles, 1988.
- G-1 W. V. Gately and R. L. Williams, "GO Methodology - Overview," NP-765 Electric Power Research Institute, (1978).
- G-2 J. N. P. Gray, "Continuous-Time Markov Methods in Solution of Practical Reliability Problems," Reliability Engineering, Vol. 11, No. 4, (1985).
- G-3 R. Ghajar and R. Billinton, "A Monte Carlo Simulation Model for the Adequacy Evaluation of Generating Systems," Reliability Engineering & System Safety, Vol. 20, No. 3, (1988).
- J-1 L. E. Johnson, "Dynamic and Steady-State Solutions for a General Availability Model," IEEE Transactions on Reliability, Vol. R-34, No. 5, (December 1985).
- J-2 K. S. Jeong, S. H. Chang, and T. W. Kim, "Development of the Dynamic Fault Tree Using Markovian Process and Super Component," Reliability Engineering, Vol. 19, No. 2 (1987).
- K-1 T. Kohda and E. J. Henley, "On Digraphs, Fault Trees, and Cut Sets," Reliability Engineering & System Safety, Vol. 20, No. 1, (1988).
- K-2 H. Kumamoto, T. Tanaka, and K. Inoue, "A New Monte Carlo Method for Evaluating System-Failure Probability," IEEE Transactions on Reliability, Vol. R-36, No. 1, (April 1987).
- L-1 N. Limnios, "A Note on the Computation of Markovian Systems," Reliability Engineering & System Safety, Vol. 23, No. 3, (1988).
- L-2 E. E. Lewis and T. Zhuguo, "Monte Carlo Reliability Modeling by Inhomogeneous Markov Processes," Reliability Engineering, Vol. 16, No. 4, (1986).
- L-3 M. O. Locks, "Recent Developments in Computing of System-Reliability," IEEE Transactions on Reliability, Vol. R-34, No. 5, (December 1985).
- L-4 A. M. Law and C. S. Larmey, An Introduction to Simulation Using SIMSCRIPT II.5, CACI, Inc.-Federal, Los Angeles, 1984.
- M-1 N. J. McCormick, Reliability and Risk Analysis, Academic Press, Inc., Orlando, Florida, 1981.
- M-2 T. Matsuoka and M. Kobayashi, "GO-FLOW: A New Reliability Analysis Methodology," Nuclear Science and Engineering, Vol. 98, No. 1, (January 1988).
- M-3 T. Matsuoka and M. Kobayashi, "The GO-FLOW Methodology: A Reliability Analysis of the Emergency Core Cooling System of a Marine Reactor Under Accident Conditions," Nuclear Technology, Vol. 84, No. 3, (March 1989).

- O-1 R. M. O'Keefe, "The Three-phase Approach: A Comment on 'Strategy-related Characteristics of Discrete-event Languages and Models'," Simulation, Vol. 47, No. 5, (November 1986).
- P-1 A. Pages and M. Gondran, System Reliability - Evaluation and Prediction in Engineering, North Oxford Academic Publishers Ltd., 1986.
- P-2 D. B. Parkinson, "Fast Availability Simulation," Reliability Engineering, Vol. 18, No. 3, (1987).
- P-3 I. A. Papazoglou and E. P. Gyftopoulos, "Markovian Reliability Analysis Under Uncertainty with an Application on the Shutdown System of the Clinch River Breeder Reactor," NUREG /CR-0405, September 1978.
- P-4 W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, Numerical Recipes, Cambridge University Press, New York, 1986.
- R-1 E. C. Russell, Building Simulation Models with SIMSCRIPT II.5, CACI, Inc.-Federal, Los Angeles, 1983.
- S-1 J. W. Schmidt and R. E. Taylor, Simulation and Analysis of Industrial Systems, Irwin, Homewood, Illinois, 1970.
- S-2 A. F. Seila, "SIMTOOLS: A Software Tool Kit for Discrete Event Simulation in Pascal," Simulation, Vol. 50, No. 3, (March 1988).
- W-1 WASH 1400, "An Assessment of Accident Risks in U.S. Commercial Nuclear Power Plants," Reactor Safety Study, WASH 1400, U.S.A.E.C., 1974.

Appendix A

DYMCAm Input File Description

Figure A.1 shows an example listing of an input file for the DYMCAm program. Line numbers are indicated to aid in describing the setup of an input file for a specific program, since different problems will require different numbers of input data file lines. This discussion should provide all information necessary to create a file to solve any specific problem for the desired system unavailability information. Any text editor can be used to create the input file, and the file can be given any name acceptable by DOS requirements.

Line 1 is the title line and can be up to 80 characters long. If the title is less than 80 characters long, it will be necessary to enter spaces to extend the line to the full length. The read statements in the Input routine are formatted reads, and therefore, if 80 characters are not found on the first line, the program will look on the next line for the remaining characters of the title, thus misreading the desired input data contained in later lines. Some text editors, such as K-EDIT, do not save the trailing blank spaces and thus could cause a problem if attempts are made to use them to create input files. One trick that can be used if the title is short, is simply to enter spaces out to column 80 of line 1, and then enter a character in column 81. K-EDIT will save the entire line, but DYMCAm will only read the first characters of the line, thus printing only the title which was desired.

Line 2 contains the number of simulated hours for which the program is to be run. The format is d(10,2) which means the program is looking for a decimal number with two digits after the decimal point, and that the number will be found in the first ten columns of line 2. For this particular

| <u>LINE NUMBER</u> | <u>INFORMATION</u> | | | | |
|--------------------|-------------------------|---------|-----------|---|---|
| 1 | Test Simulation Program | | | | |
| 2 | 1000.00 | | | | |
| 3 | 0 | | | | |
| 4 | 1000 | | | | |
| 5 | 11 | | | | |
| 6 | 0 | | | | |
| 7 | 0.00 | | | | |
| 8 | 100.00 | | | | |
| 9 | 200.00 | | | | |
| 10 | 300.00 | | | | |
| 11 | 400.00 | | | | |
| 12 | 500.00 | | | | |
| 13 | 600.00 | | | | |
| 14 | 700.00 | | | | |
| 15 | 800.00 | | | | |
| 16 | 900.00 | | | | |
| 17 | 1000.00 | | | | |
| 18 | 2 | | | | |
| 19 | BATTERY | passive | operating | 1 | 1 |
| 20 | 0.1 | 0.0 | | | |
| 21 | 1.0 | 1.0 | 0.0 | | |
| 22 | | system | process | 0 | |
| 23 | | SWITCH | process | 1 | |
| 24 | SWITCH | switch | open | 3 | 1 |
| 25 | 0.3 | 0.0 | | | |
| 26 | 1.0 | 1.0 | 0.0 | | |
| 27 | | system | command | 0 | |
| 28 | | system | power | 1 | |
| 29 | | BATTERY | process | 0 | |
| 30 | | system | process | 0 | |
| 31 | | standby | | | |
| 32 | | 1 | | | |
| 33 | | 3 | | | |
| 34 | | 100.00 | 0 | | |
| 35 | | 1 | | | |
| 36 | system | BATTERY | process | | |
| 37 | | 1 | | | |
| 38 | | 500.00 | 0 | | |
| 39 | | 1 | | | |
| 40 | system | SWITCH | command | | |
| 41 | | -1 | | | |
| 42 | | 900.00 | 1 | | |
| 43 | | SWITCH | | | |
| 44 | | open | | | |
| 45 | | 0 | | | |

Figure A.1
Example DYMCAm Input File

format specification it is not necessary to have the value right justified in the ten column field of interest. The value can be entered left justified, if desired, and the program will read all digits to the left of the decimal point as an integer value and then read the next two digits following the decimal and ignore any other characters which may be in the first ten columns. It is critical only that the decimal appear somewhere in the first 10 columns so that format specifications are satisfied. If the decimal appears in columns 9 or 10, then the one or two digits following the decimal for which values have not been assigned, will be recorded as being zero. This is true for any number read with a $d(x,y)$ format. Regardless of the value of y , as long as a decimal is somewhere in the x columns specified, then y characters will be assigned following the decimal. If y characters are present in the input field, then they will be entered, if not, then zeroes will be entered for the remaining digits. For the example shown, the input value of simulation time is 1000 hours.

Line 3 is an integer value as must be entered in column 10. The value which may be entered is either a 0 or a 1. The 0 entry signifies that the run is to be a normal run. The 1 entry indicates that the run will be a test run to see if proper program operation is occurring. Entering a 1 will cause all components to fail at their mean failure time (one over the failure rate) and all repairs to occur at their mean repair time. Thus by entering a 1, it is possible to check and make sure that all components are failing and being repaired as expected. The example shown in Figure A.1 has a 0 entered indicating the run will be a normal run.

Line 4 indicates the number of Monte Carlo trials to be performed. The number is entered as an integer value and must be right justified so that the right most character of the number is entered in column 10 of line 4. The

example shows a value of 1000 indicating that 1000 Monte Carlo trials are to be performed.

Line 5 specifies the number of time points for which dynamic system unavailability data is required. This number is also an integer value and must also be right justified with the one's digit falling in column 10. There is no requirement as to the number of time points to be entered. If desired, a zero can be entered and no dynamic information will be calculated for the system. For the example problem, 11 time points will be used for the dynamic unavailability analysis.

Line 6 is an integer value referring to the type of time distribution desired for the dynamic unavailability analysis. The integer number 0, 1, or 2 must be entered in column 10 of line 6. Entering a 0 indicates that the next lines of the input file will contain the desired time points. For the example of Figure A.1, a value of 0 is specified, indicating that the next 11 (number of time points specified in line 5) lines of the input line will contain the time points of interest. If a 1 had been entered, then the 11 time points would have been chosen as uniformly distributed between time zero and the value specified in line 2. The program automatically chooses zero and the value of line 2 as two of its time points, thus the remaining 9 points for this example would be chosen uniformly distributed between the beginning and end times. For this case, the points chosen would be the same ones entered in lines 7 through 17, therefore this program feature allows for simplification of the input file.

If a value of 2 is entered in column 10 of line 6, then the program will choose values for the time points which are log distributed between the zero time and the end of simulation time specified in line 2. As was the case with entering a 1, the time zero point and the end of simulation time

are automatically chosen. The remaining $N - 2$ points are calculated by taking the log of the line 2 value, dividing it by one less than the value of line 5, and then taking the inverse log of integer multiples of this result, for time points between the two end times of the distribution. This feature may be useful for evaluating the unavailability of a system which is suspected of having an exponentially distributed result. Since the time required to run the simulation program is directly related to the number of time the program is interrupted to take another time dependent unavailability sample, it is desireable to keep the number of time points specified in the input file to a minimum, while still providing sufficient data to properly evaluate the dynamic behavior of the system.

Line 18 of the input file specifies the number of components contained in the system. For the example the number of components is 2. This value will always be an integer value and must be entered right justified with the right most digit falling in column 10. For every component indicated by this number, there will be a minimum of five line of data in the input file. For the example of Figure A.1, the first component is described in lines 19 through 23 and the second component is described in lines 24 through 30.

Each component must have a first line entered in the format of lines 19 and 24. The first 10 columns are reserved for the components name. The name can contain any characters desired, but must not contain spaces. It need not be left or right justified. It need only be less than or equal to 10 characters in length. The SIMSCRIPT language distinguishes between small and capital letters, therefor it is important that if capital letters are used for component names, that this is done consistently every where a specific component name is mentioned. All other text, other than component names, must be entered in lower case letters, since this is what the DYMCA

program has been programed to recognize.

Columns 11 through 20 for the first line of each component must contain the component type designation. This, as all text, need not be justified, but must be in lower case letters. Columns 21 through 30 should contain the components initial state upon execution of the simulation. This information must also be in lower case letters. Also on this line, the number of input and output signals used by the component should be specified. Any number of input and output signals can be assigned to a given component, however, for all components, at least one input and one output signal must exist. The number of inputs is an integer value and must be right justified in the column 31 to 35 field, while the number of output signals must be entered as an integer value right justified in the 36 to 40 column field. Line 19 of the example refers to a passive element named BATTERY which is initially in standby at time zero and has one input and one output signal. Line 24 indicates a switch named SWITCH which is initially open and has three inputs and one output signal.

The second line of each component data field (lines 20 and 25 of the example) contains the failure data for the component. The first 10 columns contain the demand failure probability. The format for reading this value is d(10,5). As discussed earlier, this means that the data will be contained in a field 10 columns wide, and may include five digits following the decimal. If more than five digits are entered after the decimal, they will be ignored. The second data field of this line is from column 11 to column 20. This will contain the failure rate, lambda, for the component. The format for this value is also d(10,5). As stated above, neither of these values need be justified in their data fields. It is only critical that a decimal point be entered somewhere in the field. Line 20 of Figure A.1, for

example, indicates a value of 0.1 for the BATTERY demand failure probability. This value was entered with only one decimal place, since only one decimal was needed, regardless of the fact that the format specified five decimal places can be entered. Likewise for the SWITCH in line 25, and the failure rates for both components.

The third line for each component (lines 21 and 26) must contain the repair information. Three data values are entered and each is read in the d(10,5) format. The first value is the alpha parameter for the Weibull distribution and it must be found in columns 1 through 10. The second value is the beta parameter for the Weibull repair distribution and must be entered in columns 11 through. The third value is the probability the component is repairable once it has failed. This number is entered in columns 21 through 30. If exponentially distributed repair is to be considered, this can be accomplished by entering a 0 for the value of alpha, and treating beta as being equal to the mean repair time for the component (one over mu, the exponential repair rate). For cases when a 1 is entered in line 3 of the input file, the mean time to repair is treated as being equal to the Weibull parameter, beta, regardless of the value of the alpha parameter. For the example shown, repair is not considered, thus the values entered in lines 21 and 26 do not have physical significance, except for the zeros, which simply indicate that once the component fails, it stays failed since it is not repairable.

For every signal in the system, a line like lines 22, 23, and 27 through 30 must be specified. Since signals must be associated with the components they link, they will always be listed following the component. The number of signals described following any component will equal the sum of the number of input and output signals specified for the given component.

For the example shown, the BATTERY has one input and one output, thus two signals are specified. For the SWITCH, there are three inputs and one output, thus four signals are specified. The input signals for a component must always be specified first and the output components last. The order of specifying several input or output files for a given component, however, is not important as long as the above rule is obeyed. Every signal which does not originate from, or terminate at the system level, must be contained in two component listings. This is clearly evident because each signal must have an origin and a destination. Thus if it does not come from or go to the system, it must travel between two components of the system.

Information concerning signals must always begin in the column 11 to 20 field. The first 10 columns to provide ease in viewing the file. The first field of the description (columns 11 to 20) attaches the signal to another component. For input signals, this field contains the name of where the signal came from (either the system or an other component), and for output signals the data field contains the destination of the signal (either the system or an other component). Thus each signal is tied between two components.

The second data field for each component is contained in columns 21 through 30 and indicates the type of signal (either command, power, or process). As with all text data fields, the data need not be justified. The third piece of data concerning each signal, is it's strength at the start of the simulation. For power and process signals the strength is 0 if power is not available or the process variable is not present, and the strength is 1 if power is available or the process variable exists. For command signals, a value of 0 indicates no command, while a value of 1 indicates to open the switch or valve (or start the active component). A value of -1 indicates to

close the valve or switch (or to stop the active component). These values are entered as integers and are right justified in column 35. For the example of Figure A.1, the BATTERY has one input and one output signal. The input is a process signal coming from the system and is initially off, while the output signal is a process signal going to the SWITCH and is also initially off. The SWITCH has three inputs and one output. Two of the inputs are from the system and reflect the power and command signals to the SWITCH. Initially the switch has power but no command signal. The other input to the switch is the process signal which comes from the BATTERY. The output signal is a process signal which goes to the system.

Line 31 provides information about the initial state of the system. The program does not calculate the system state until a time 0+ which is slightly greater than time zero, thus to artificially set the system to its desired initial operating state it is necessary to set it at the beginning of the run. For the system to be available at time zero, the system status is set to operating or standby. Thus the value entered for initial system state is either operating, standby, or failed. This data is entered in the first 10 columns of the input file line. Line 31 of the example indicates the system initially starts in the standby condition.

The next required line in the input file is the system success criteria. This is the number of output signals directed to the system which must be on in order for the system to be considered available. It is entered as an integer value and must be right justified in column 10 of the data line. For the example, the value entered in line 32 is one, specifying that at least one output signal to the system must be on in order for the system to be available. For this example, there is only one output signal to the system, the output process signal from the switch, thus the system is only

available if the switch is closed and an output process signal is being generated, i.e. the BATTERY must also be operating.

Next, the number of external events to be included in the problem scenario must be entered. This value will be an integer and is read right justified from column 10 of the data file line. This value may be zero if the problem to be analyzed is not a phased mission one, and if this is the case, this will be the last line of the input file. For the example of Figure A.1, line 33 indicates that there are 3 external events for this problem.

For each external event, at least four lines of data must be entered. The first line contains the time at which the event is scheduled to occur. This information is contained in columns 1 to 10 and is read in the d(10,2) format. Following this, in columns 11 to 20, the number of components effected by the external event are given. This is an integer value and must be entered right justified in column 20. Every external event must affect at least one component or signal, but not necessarily both, therefor this value may often be 0 as it is in lines 34 and 38 of the example. If the value is 1 or greater, then the next lines will list the components effected by the external event. Each line, like line 43 of the example, simply lists the name of the affected component. The name must be found in the first 10 columns of the data file line. For the example, the external event changes the state of the SWITCH. The program is written such that all components changed by a given external event, are affected in the same manner. Thus the next data file line following the component names, gives the new state of these components. For the example, the external event opens the SWITCH at 900.00 hours into the simulation. Thus line 44 contains the instruction to open. This component change of state must be entered in the first 10 columns

of the data line.

The next line of an external event specifies the number of signals affected by the event. This will be an integer value and must be entered right justified in column 10 of the data line. For the example of Figure A.1, the third external event does not change any signals as is indicated by the 0 in line 45. The first two external events change one signal each. This is indicated in lines 35 and 39 of the example input file. If a signal is changed, then two lines must be entered for each signal changed by the external event. The first line contains the origin of the signal, the destination of the signal, and the type of signal. These three data entries are text information and are entered in columns 1 to 10, 11 to 20, and 21 to 30 respectively. The next input data line contains the new strength of the signal. This will be an integer value and is entered right justified in column 10 of the data file line. For the example of Figure A.1, the first external event changes the process signal from the system to the BATTERY (line 36). The new strength (line 37) specifies that the signal is to be turned on so that the BATTERY may now supply current. The second external event of the example effects the command signal from the system to the SWITCH. It causes the command signal to change to -1 at the 500.00 hour time point which will cause the switch to close. provided it does not experience a demand failure. Line 40 of the example specifies the signal, while line 41 gives the new value.

With the current program structure, it is possible to change many signals with a single external event, and to change each to a different signal strength. These same signals may be changed again at a later time in the simulation by another external event. Components, on the other hand, can only be changed once by an external event. This means that if an external

event is used at the 500.00 hour time point to open a switch, the same switch can not be closed with an external event at a later time in the simulation (although it may have its input command signal changed). This is because of the way external events were treated in development of this basic demonstration program. It would be possible to modify the program to allow multiple state changes of a given component, if such a capability were desireable.

Also with the current structure, all components changed by a given external event must be changed to the same new state. This is not such a problem since any number of external events can be scheduled to occur at exactly the same time. In fact, the motivating idea for the external event was that each event would effect only a single component or type of component. If it is desireable, the External Event routine could certainly be modified to allow multiple component changes during a single external event.

This appendix should supply all the information necessary for writing input files for the DYMCAW program. Care must be taken to ensure that all information is properly formatted. For further examples of input files, Appendix D can be consulted which contains several input files used for the various test runs performed in chapters four and five. Also note in Appendix D that all data file lines (with the exception of the title line) contain data only up through column 40. Since SIMSCRIPT will not look beyond this point for any data, it is possible to use this "blank space" to include comments concerning the input file data for future reference and ease of understanding. This has been done for all test cases run.

Appendix B
DYMCAM Program Listing

```
1 preamble
2 ''
3 ''    RISK - Test program to simulate system behavior
4 ''
5 ''    03/28/89
6 ''
7     permanent entities
8
9      every component.record
10         has a component_name,
11             a component_type,
12             a number_inputs,
13             a number_outputs,
14             a response_function,
15             an initial_state,
16             a demand_failure_frequency,
17             a run_failure_frequency,
18             a repair_probability,
19             a repair_function_shape, and
20             a repair_function_scale
21
22      every external.event.record
23         has an occurrence_time,
24             a number_components,
25             a new_state,
26             a number_signals, and
27             a new_strength
28
29      define response_function as a subprogram variable
30      define component_name, component_type, initial_state,
31          and new_state as text variables
32      define demand_failure_frequency, run_failure_frequency,
33          repair_probability, repair_function_shape,
34          and repair_function_scale as real variables
35      define number_inputs, number_outputs, number_components,
36          number_signals, and new_strength as integer variables
37 ''
38 ''    2-d arrays associated with permanent entities.
39 ''
40      define input.name, output.name, input.signal.type,
41          output.signal.type, extevnt.component, extevnt.origin,
42          extevnt.destination, and extevnt.stype
43          as 2-dimensional text arrays
44      define input.signal.strength and output.signal.strength
45          as 2-dimensional integer arrays
46      define test as a 1-dimensional text array
47      define signal.status as a 1-dimensional integer array
48
49      processes include call.update, schedule.avail.samples,
50          schedule.external.events, repair.supervisor,
51          stop.tank, and stop.scenario           ''TANK
52
53      every component
54         has a name,
55             a component.type,
```

```

56      a response.function,
57      an old.state,
58      a state,
59      a demand.failure.frequency,
60      a run.failure.frequency,
61      a repair.probability,
62      a repair.function.shape,
63      a repair.function.scale,
64      a failure.time,
65      a status,
66      and owns an input.sset and           ''TANK
67      an output.sset
68      and may belong to a system.cset,      ''TANK
69      a tank.input.cset,
70      a tank.output.cset,                  ''TANK
71      and an extevnt.cset
72
73      every tank                         ''TANK
74          has a high.level,             ''TANK
75          a low.level,                ''TANK
76          a high.set,                 ''TANK
77          a low.set,                  ''TANK
78          a level,                   ''TANK
79          a flow.rate.in,            ''TANK
80          a flow.rate.out,           ''TANK
81          a net.flow.rate,          ''TANK
82          and owns a tank.input.cset, ''TANK
83          a tank.output.cset,        ''TANK
84          a tank.input.sset,         ''TANK
85          a tank.output.sset,        ''TANK
86          and belongs to a system.tset   ''TANK
87
88      every external.event
89          has an occurrence.time,
90          a new.state,
91          a number.signals,
92          a signal.origin,
93          a signal.destination,
94          a signal.typee, and
95          a new.strength
96          and owns an extevnt.cset
97          and belongs to a system.eset
98
99      every availability
100         has a time.avail, and
101         a time.avail.data
102
103     define time avail as a 1-dimensional real array
104     define time.avail and time.avail.data as real variables
105     define tank.condition as an integer function           ''TANK
106     define response.function as a subprogram variable
107     define name, component.type, old.state, state, new.state,
108         signal.origin, signal.destination, and signal.typee
109         as text variables
110     define demand.failure.frequency, run.failure.frequency,

```

```
111      repair.probability, repair.function.shape,
112      repair.function.scale, failure.time, occurrence.time,
113      high.level, low.level, high.set, low.set,          ''TANK
114      flow.rate.in, flow.rate.out, net.flow.rate,        ''TANK
115      and number.signals as real variables
116      define status and new.strength as integer variables
117      define level as a continuous double variable       ''TANK
118  ''
119  ''    Later versions may define signals as processes (so time delays
120  ''    can be built in).
121  ''
122 temporary entities
123
124     every signal
125         has a signal.type,
126         an origin,
127         a destination,
128         an old.strength, and
129         a strength
130         and may belong to an output.sset,
131         an input.sset,
132         a tank.input.sset,                      ''TANK
133         a tank.output.sset,                     ''TANK
134         a system.boundary.sset,
135         a system.success.sset, and
136         a system.sset
137
138     define cptr, sptr, eptr, aptr, and tptr           ''TANK
139         as 1-dimensional pointer arrays
140
141     define signal.type, origin, and destination as text variables
142     define old.strength and strength as integer variables
143  ''
144  ''    System characteristics.
145  ''
146     the system owns a system.boundary.sset,
147         a system.success.sset,
148         a system.cset,
149         a system.sset,
150         a system.eset, and
151         a system.tset                           ''TANK
152
153     define failure.translation as a text function
154     define job.title, initial.system.state, and system.state
155         as text variables
156     define system.ind.var and simulation.time as real variables
157     define ntrial, system.success.criterion, ntimes,
158         distribution.type, run.type, and total.signal.count
159         as integer variables
160     define unavailability.dist as a 1-dimensional real array
161     define trial.unavail as a real variable
162
163     accumulate trial.availability as the mean of system.ind.var
164     tally average.unavailability as the mean,
165         variance.unavailability as the variance,
```

```
166      maximum.unavailability as the maximum,
167      and minimum.unavailability as the minimum of
168      trial.unavail
169
170      define .off to mean 0
171      define .on to mean 1
172      define .no to mean 0
173      define .yes to mean 1
174      define .working to mean 1
175      define .resetting to mean 2
176      define .awaiting.repair to mean 3
177      define .under.repair to mean 4
178      define .not.repairable to mean 5
179      define .reset.run to mean 6
180
181 end ''preamble
```

```
1 main
2     define trial as an integer variable
3 ''
4 ''    Problem input
5 ''
6     call input
7     call run.initialize
8     call tank.initialize.run                                ''TANK
9     add .003 to simulation.time
10    for trial = 1 to ntrial
11    do
12        call trial.initialize
13        call tank.initialize.trial                          ''TANK
14        activate a call.update now
15        activate a schedule.avail.samples now
16        activate a schedule.external.events now
17        activate a stop.tank in simulation.time hours      ''TANK
18        activate a stop.scenario in simulation.time hours
19        start simulation
20        let unavailability.dist(trial) = 1 - trial.availability
21        let trial.unavail = trial.availability            ''TANK
22        let time.v = 0
23        reset totals of system.ind.var
24    loop
25
26    call run.output
27
28 end ''main
```

```

1 routine active given component
2 ''
3 '' Develops output signals for an active component
4 '' using explicit command signals. Assumes that the component
5 '' has one or more command signal inputs, power inputs, and
6 '' process inputs:
7 ''
8 ''      input command --- |_____
9 ''      input power   --- |_____| --- output process
10 ''     input process  --- |_____|

11 ''
12 '' Condensed decision table:
13 ''
14 ''      Command    Power   Process   Initial   Final    Process
15 ''      Case       Input   Input     Input     State    State   Output
16 ''      ----      -----  -----  -----  -----
17 ''      1         -       -        -       failed  failed  no
18 ''      2         -       no       -       standby standby no
19 ''      3       stop     yes      -       standby standby no
20 ''      4      none     yes      -       standby standby no
21 ''      5      start    yes     no       standby standby* no
22 ''          failed
23 ''      6      start    yes     yes      standby standby* no
24 ''          operating
25 ''      7         -       no       -       operating standby no
26 ''      8       stop     yes     no       operating failed  no
27 ''          standby
28 ''      9       stop     yes     yes      operating operating* yes
29 ''          standby no
30 ''      10      none    yes     no       operating failed  no
31 ''      11      none    yes     yes      operating operating yes
32 ''      12      start   yes     no       operating failed  no
33 ''      13      start   yes     yes      operating operating yes
34 ''      14         -       -        -       standby* standby* no
35 ''      15         -       no       -       operating* operating* no
36 ''      16         -       yes     no       operating* failed  no
37 ''      17         -       yes     yes      operating* operating* yes
38 ''
39      define rule as a saved 2-dimensional text array
40      define component as a pointer variable
41      define index.command, total.command, number.power, total.power,
42          number.process, total.process, output.strength, ruletype,
43          success, and j as integer variables
44      define later.case as a saved integer variable
45 ''
46 ''      Enter decision table.
47 ''
48      if later.case eq .no
49          reserve rule as 17 by 4
50          let rule(1,1) = ""      let rule(1,2) = ""
51          let rule(1,3) = ""      let rule(1,4) = "failed"
52          let rule(2,1) = ""      let rule(2,2) = "no"
53          let rule(2,3) = ""      let rule(2,4) = "standby"
54          let rule(3,1) = "stop"  let rule(3,2) = "yes"
55          let rule(3,3) = ""      let rule(3,4) = "standby"

```

```
56      let rule(4,1) = "none"      let rule(4,2) = "yes"
57      let rule(4,3) = ""          let rule(4,4) = "standby"
58      let rule(5,1) = "start"     let rule(5,2) = "yes"
59      let rule(5,3) = "no"        let rule(5,4) = "standby"
60      let rule(6,1) = "start"     let rule(6,2) = "yes"
61      let rule(6,3) = "yes"       let rule(6,4) = "standby"
62      let rule(7,1) = ""          let rule(7,2) = "no"
63      let rule(7,3) = ""          let rule(7,4) = "operating"
64      let rule(8,1) = "stop"      let rule(8,2) = "yes"
65      let rule(8,3) = "no"        let rule(8,4) = "operating"
66      let rule(9,1) = "stop"      let rule(9,2) = "yes"
67      let rule(9,3) = "yes"       let rule(9,4) = "operating"
68      let rule(10,1) = "none"     let rule(10,2) = "yes"
69      let rule(10,3) = "no"        let rule(10,4) = "operating"
70      let rule(11,1) = "none"     let rule(11,2) = "yes"
71      let rule(11,3) = "yes"       let rule(11,4) = "operating"
72      let rule(12,1) = "start"     let rule(12,2) = "yes"
73      let rule(12,3) = "no"        let rule(12,4) = "operating"
74      let rule(13,1) = "start"     let rule(13,2) = "yes"
75      let rule(13,3) = "yes"       let rule(13,4) = "operating"
76      let rule(14,1) = ""          let rule(14,2) = ""
77      let rule(14,3) = ""          let rule(14,4) = "standby*"
78      let rule(15,1) = ""          let rule(15,2) = "no"
79      let rule(15,3) = ""          let rule(15,4) = "operating*"
80      let rule(16,1) = ""          let rule(16,2) = "yes"
81      let rule(16,3) = "no"        let rule(16,4) = "operating*"
82      let rule(17,1) = ""          let rule(17,2) = "yes"
83      let rule(17,3) = "yes"       let rule(17,4) = "operating*"
84      let later.case = .yes
85      always
86  ''
87  ''  Determine input signal status. Assume that "start" and "stop"
88  ''  commands cancel each other out (respective values of 1 and -1).
89  ''
90  for every signal in input.sset(component)
91  do
92      if signal.type(signal) eq "process"
93          add 1 to total.process
94          if strength(signal) eq .on
95              add 1 to number.process
96          always
97      else
98          if signal.type(signal) eq "power"
99              add 1 to total.power
100             if strength(signal) eq .on
101                 add 1 to number.power
102             always
103         else
104             add 1 to total.command
105             add strength(signal) to index.command
106         always
107     always
108   loop
109  ''
110 ''  Develop test vector for comparison with rules. Assume that
```

```
111 '' a single process signal is sufficient, and that a single power
112 '' signal is sufficient (i.e., OR gates).
113 ''
114     if index.command eq -1
115         let test(1) = "stop"
116     else
117         if index.command eq 0
118             let test(1) = "none"
119         else
120             let test(1) = "start"
121         always
122     always
123     if number.power ge 1
124         let test(2) = "yes"
125     else
126         let test(2) = "no"
127     always
128     if number.process ge 1
129         let test(3) = "yes"
130     else
131         let test(3) = "no"
132     always
133     let test(4) = state(component)
134 ''
135 '' Determine appropriate rule.
136 ''
137     for ruletype = 1 to 17
138     do
139         for j = 1 to 4
140         do
141             if rule(ruletype,j) ne "" and rule(ruletype,j) ne test(j)
142                 go to 'next'
143             always
144             loop
145             go to 'found'
146         'next'
147         loop
148 ''
149 '' Select rule.
150 ''
151 'found'
152 select case ruletype
153
154 case 1, 16
155     let state(component) = "failed"
156     let output.strength = .no
157
158 case 2, 3, 4, 7
159     let state(component) = "standby"
160     let output.strength = .no
161
162 case 5
163     call demand.test giving component yielding success
164     if success eq .no
165         let state(component) = "standby*"
```

```
166      let output.strength = .no
167      else
168          let state(component) = "failed"
169          let output.strength = .no
170      always
171
172  case 6
173      call demand.test giving component yielding success
174      if success eq .no
175          let state(component) = "standby*"
176          let output.strength = .no
177      else
178          let state(component) = "operating"
179          let output.strength = .yes
180      always
181
182  case 8
183      call demand.test giving component yielding success
184      if success eq .no
185          let state(component) = "failed"
186          let output.strength = .no
187      else
188          let state(component) = "standby"
189          let output.strength = .no
190      always
191
192  case 9
193      call demand.test giving component yielding success
194      if success eq .no
195          let state(component) = "operating*"
196          let output.strength = .yes
197      else
198          let state(component) = "standby"
199          let output.strength = .no
200      always
201
202  case 10, 12
203      let state(component) = "failed"
204      let output.strength = .no
205
206  case 11, 13
207      let state(component) = "operating"
208      let output.strength = .yes
209
210  case 14
211      let state(component) = "standby*"
212      let output.strength = .no
213
214  case 15
215      let state(component) = "operating*"
216      let output.strength = .no
217
218  case 17
219      let state(component) = "operating*"
220      let output.strength = .yes
```

```
221      default
222  ''    Error messages can be put here if rule not matched.
223  ''    endselect
224  ''    Update output signals.
225  ''    for every signal in output.sset(component)
226      let strength(signal) = output.strength
227
228  return
229
230 end ''active
```

```
1 process availability
2 ''
3 ''    This process totals the sum of the system indicator
4 ''    variable at the specified time points. At the completion
5 ''    of all trials the totals are divided by the number of
6 ''    trials to determine the time dependent system availability.
7 ''
8    while time.v lt (simulation.time + 10)
9      do
10        suspend
11        add system.ind.var to time.avail.data(availability)
12      loop
13
14    suspend
15
16 end ''availability
```

```
1 process call.update
2 ''
3 ''    This should be a process to keep the process component
4 ''    from destroying itself when it tries to call a system
5 ''    update.
6 ''
7     while time.v lt .000004
8     do
9         wait .000005 hours
10        for every component in system.cset
11        do
12            resume the component
13        loop
14        for every tank in system.tset
15        resume the tank
16        wait .0005 hours
17        for i = 1 to dim.f(cptr(*))
18        do
19            if component.type(cptr(i)) eq "active"
20                or component.type(cptr(i)) eq "passive"
21                if state(cptr(i)) ne "operating"
22                    interrupt the component called cptra(i)
23                always
24                always
25            loop
26        loop
27        call system.update
28
29    return
30
31 end ''call.update
                                ''TANK
                                ''TANK
```

```

1 routine check.valve given component
2 ''
3 ''    Develops output signals for a check valve.
4 ''
5 ''
6 ''          input process --- [ ] --- output process
7 ''
8 ''
9 ''    Condensed decision table:
10 ''
11 ''      Process      Initial      Final       Process
12 ''      Case   Input     State       State      Output
13 ''      ----  -----  -----  -----  -----
14 ''      1      -      failed_closed  failed_closed  no
15 ''      2      no      closed        closed        no
16 ''      3      yes     closed        failed_closed  no
17 ''          -          -          open          yes
18 ''      4      no      failed_open   failed_open   no
19 ''      5      yes     failed_open   failed_open   yes
20 ''      6      no      open          failed_open   no
21 ''          -          -          closed        no
22 ''      7      yes     open          open          yes
23 ''
24 define rule as a saved 2-dimensional text array
25 define component as a pointer variable
26 define number.process, total.process, output.strength,
27     ruletype, success and j as integer variables
28 define later.case as a saved integer variable
29 ''
30 ''    Enter decision table.
31 ''
32 if later.case eq .no
33     reserve rule as 7 by 2
34     let rule(1,1) = ""
35     let rule(2,1) = "no"
36     let rule(3,1) = "yes"
37     let rule(4,1) = "no"
38     let rule(5,1) = "yes"
39     let rule(6,1) = "no"
40     let rule(7,1) = "yes"
41     let later.case = .yes
42 always
43 ''
44 ''    Determine input signal status.
45 ''
46 for every signal in input.sset(component)
47 do
48     if signal.type(signal) eq "process"
49         add 1 to total.process
50         if strength(signal) eq .on
51             add 1 to number.process
52         always
53     always
54 loop
55 ''

```

```
56 '' Develop test vector for comparison with rules. Assume that
57 '' a single process signal is sufficient (i.e., an OR gate).
58 ''
59     if number.process ge 1
60         let test(1) = "yes"
61     else
62         let test(1) = "no"
63     always
64     let test(2) = state(component)
65 ''
66 '' Determine appropriate rule.
67 ''
68 for ruletype = 1 to 7
69 do
70     for j = 1 to 2
71         do
72             if rule(ruletype,j) ne "" and rule(ruletype,j) ne test(j)
73                 go to 'next'
74             always
75             loop
76             go to 'found'
77         'next'
78     loop
79 ''
80 '' Select rule.
81 ''
82 'found'
83 select case ruletype
84
85 case 1
86     let state(component) = "failed_closed"
87     let output.strength = .no
88
89 case 2
90     let state(component) = "closed"
91     let output.strength = .no
92
93 case 3
94     call demand.test giving component yielding success
95     if success eq .no
96         let state(component) = "failed_closed"
97         let output.strength = .no
98     else
99         let state(component) = "open"
100        let output.strength = .yes
101    always
102
103 case 4
104     let state(component) = "failed_open"
105     let output.strength = .no
106
107 case 5
108     let state(component) = "failed_open"
109     let output.strength = .yes
110
```

```
111    case 6
112        call demand.test giving component yielding success
113        if success eq .no
114            let state(component) = "failed_open"
115            let output.strength = .no
116        else
117            let state(component) = "closed"
118            let output.strength = .no
119        always
120
121    case 7
122        let state(component) = "open"
123        let output.strength = .yes
124
125    default
126 ''
127 ''    Error messages can be put here if rule not matched.
128 ''    endselect
129 ''
130 ''    Update output signals.
131 ''    for every signal in output.sset(component)
132        let strength(signal) = output.strength
133
134    return
135
136
137
138 end ''check.valve
```

```
1 process component
2 ''
3 ''    Tracks behavior of all components after initial demand (change).
4 ''    Includes repair.  Uses exponential failure time model.
5 ''
6     define mean.failure.time, default.time, e1, and
7         e2 as real variables
8
9     'term'
10    suspend
11    while time.v lt (simulation.time + 10)
12    do
13        'reset'
14        let status(component) = .working
15        if run.failure.frequency(component) gt 0
16            let mean.failure.time = 1./run.failure.frequency(component)
17            if run.type eq 1
18                wait mean.failure.time hours
19                go to 'repair'
20            otherwise
21                wait exponential.f(mean.failure.time,1) hours
22            'repair'
23            if status(component) eq .resetting
24                go to 'reset'
25            always
26            if status(component) eq .reset.run
27                go to 'term'
28            always
29            if state(component) eq "open" or
30                state(component) eq "closed" or
31                state(component) eq "operating"
32                let old.state(component) = state(component)
33                let state(component) = failure.translation(component)
34                activate a call.update now
35            always
36        else
37            let default.time = simulation.time + 10.0
38            wait default.time hours
39            if status(component) eq .resetting
40                go to 'reset'
41            always
42            if status(component) eq .reset.run
43                go to 'term'
44            always
45        always
46            let status(component) = .awaiting.repair
47            let failure.time(component) = time.v
48            activate a repair.supervisor now
49        suspend
50        if status(component) eq .reset.run
51            go to 'term'
52        always
53 ''
54 ''    REPAIR
55 ''
```

```
56      let status(component) = .under.repair
57      let e1 = repair.function.shape(component)
58      let e2 = repair.function.scale(component)
59      if run.type eq 1
60          wait e2 hours
61          go to 'good'
62      otherwise
63          wait weibull.f(e1,e2,1) hours
64          'good'
65          if status(component) eq .reset.run
66              go to 'term'
67          always
68          let old.state(component) = state(component)
69          select case component.type(component)
70
71          case "active", "passive"
72              let state(component) = "standby"
73
74          case "switch"
75              let state(component) = "open"
76
77          case "valve", "check.valve"
78              let state(component) = "closed"
79
80          default
81              print 1 line thus
82                  The component type was not matched in the repair routine.
83
84          endselect
85          activate a call.update now
86      loop
87
88      suspend
89
90 end ''component
```

```
1 routine demand.test given component yielding success
2 ''
3 ''    Determines if given component succeeds or fails on demand,
4 ''    using the demand.failure.frequency for the component.
5 ''
6     define component as a pointer variable
7     define success as an integer variable
8     if random.f(1) le demand.failure.frequency(component)
9         let success = .no
10    else
11        let success = .yes
12    always
13
14    return
15
16 end ''demand.test
```

```
1 process external.event
2 ''
3 ''    Schedules a change in the system (either to component status
4 ''    or signal strength) occurrence.time hours into the simulation.
5 ''
6     while time.v lt (simulation.time + 10)
7     do
8         suspend
9         for every component in extevnt.cset(external.event)
10        do
11            let old.state(component) = state(component)
12            let state(component) = new.state(external.event)
13        loop
14
15        if number.signals(external.event) eq 1
16        for j = 1 to number.signals(external.event)
17        do
18            for every signal in system.sset
19                with origin(signal) eq signal.origin(external.event)
20                and destination(signal) eq
21                    signal.destination(external.event)
22                and signal.type(signal) eq signal.typee(external.event)
23            find the first case
24            if found
25                let old.strength(signal) = strength(signal)
26                let strength(signal) = new.strength(external.event)
27                always
28            loop
29            else
30            if number.signals(external.event) ne 0
31                print 1 line thus
32                An external event was entered with more than one signal change.
33            always
34            always
35            call system.update
36        loop
37
38        suspend
39
40 end ''external.event
```

```
1 function failure.translation(component)
2 ''
3 ''    Determines status of "failed" component.
4 ''
5     define component as an integer variable
6     define mode as a text variable
7
8     select case component.type(component)
9
10    case "active", "passive"
11        let mode = "failed"
12
13    case "check.valve", "valve", "switch"
14        if state(component) eq "open"
15            let mode = "failed_closed"
16        always
17            if state(component) eq "closed"
18                let mode = "failed_open"
19        always
20            if state(component) ne "open" and
21                state(component) ne "closed"
22                print 1 line thus
23                Failure translation didn't function properly!
24        always
25
26    default
27        print 1 line thus
28        Failure translation routine rule not matched!
29
30    endselect
31
32    return with mode
33
34 end ''failure.translation
```

```
1 routine input
2 ''
3 ''    Problem input routine.
4 ''
5     define infile and outfile as text variables
6
7     write as /, "Enter DOS input file name => ",+
8     read infile
9     write as /, "Enter DOS output file name => ",+
10    read outfile
11    open 7 for input, file name = infile
12    use 7 for input
13    open 8 for output, file name = outfile
14    use 8 for output
15 ''
16 ''    Title, general characteristics.
17 ''
18     read job.title as t 80, /
19     write job.title as t 80, /
20     read simulation.time as d(10,2), /
21     write simulation.time as d(10,2), /
22     read run.type as i 10, /
23     write run.type as i 10, /
24     read ntrial as i 10, /
25     write ntrial as i 10, /
26     read ntimes as i 10, /
27     write ntimes as i 10, /
28     read distribution.type as i 10, /
29     write distribution.type as i 10, /
30     reserve time_avail(*) as ntimes
31     if distribution.type eq 0
32         for i = 1 to ntimes
33             do
34                 read time_avail(i) as d(10,2), /
35                 write time_avail(i) as d(10,2), /
36             loop
37     always
38 ''
39 ''    Component characteristics.
40 ''
41     read n.component.record as i 10, /
42     write n.component.record as i 10, /
43     create every component.record
44     reserve input.name(*,*), output.name(*,*), input.signal.type(*,*),
45         output.signal.type(*,*), input.signal.strength(*,*), and
46         output.signal.strength(*,*) as n.component.record by *
47     for i = 1 to n.component.record
48         do
49             read component_name(i),
50                 component_type(i),
51                 initial_state(i),
52                 number_inputs(i), and
53                 number_outputs(i)
54                 as 3 t 10, 2 i 5, /
55             write component_name(i),
```

```
56      component_type(i),
57      initial_state(i),
58      number_inputs(i), and
59      number_outputs(i)
60      as 3 t 10, 2 i 5, /
61      read demand_failure_frequency(i) and
62      run_failure_frequency(i)
63      as 2 d(10,5), /
64      write demand_failure_frequency(i) and
65      run_failure_frequency(i)
66      as 2 d(10,5), /
67      read repair_function_shape(i),
68      repair_function_scale(i), and
69      repair_probability(i)
70      as 3 d(10,5), /
71      write repair_function_shape(i),
72      repair_function_scale(i), and
73      repair_probability(i)
74      as 3 d(10,5), /
75  ''
76  '' Input signals for component.
77  ''
78      reserve input.name(i,*),
79          input.signal.type(i,*), and
80          input.signal.strength(i,*)
81          as number_inputs(i)
82      for j = 1 to number_inputs(i)
83      do
84          read input.name(i,j),
85          input.signal.type(i,j), and
86          input.signal.strength(i,j)
87          as b 11, 2 t 10, i 5, /
88          write input.name(i,j),
89          input.signal.type(i,j), and
90          input.signal.strength(i,j)
91          as b 11, 2 t 10, i 5, /
92          if trim.f(input.name(i,j),0) eq "system"
93              add 1 to total.signal.count
94          always
95      loop
96  ''
97  '' Output signals for components.
98  ''
99      reserve output.name(i,*),
100         output.signal.type(i,*), and
101         output.signal.strength(i,*)
102         as number_outputs(i)
103     for j = 1 to number_outputs(i)
104     do
105         read output.name(i,j),
106         output.signal.type(i,j), and
107         output.signal.strength(i,j)
108         as b 11, 2 t 10, i 5, /
109         write output.name(i,j),
110         output.signal.type(i,j), and
```

```
111          output.signal.strength(i,j)
112          as b 11, 2 t 10, i 5, /
113      loop
114      add number_outputs(i) to total.signal.count
115  loop
116  ,
117  //  System characteristics.
118  //
119  read initial.system.state as t 10, /
120  write initial.system.state as t 10, /
121  read system.success.criterion as i 10, /
122  write system.success.criterion as i 10, /
123  ,
124  //  External event records.
125  //
126  read n.external.event.record as i 10, /
127  write n.external.event.record as i 10, /
128  if n.external.event.record gt 0
129  create every external.event.record
130  reserve extevnt.component(*,*), extevnt.origin(*,*), and
131  extevnt.destination(*,*), extevnt.stype(*,*)
132  as n.external.event.record by *
133
134  for i = 1 to n.external.event.record
135  do
136      read occurrence_time(i) as d(10,2)
137      write occurrence_time(i) as d(10,2)
138      read number_components(i) as i 10, /
139      write number_components(i) as i 10, /
140      if number_components(i) gt 0
141          reserve extevnt.component(i,*) as number_components(i)
142          for j = 1 to number_components(i)
143          do
144              read extevnt.component(i,j) as t 10
145              write extevnt.component(i,j) as t 10
146          loop
147          read new_state(i) as /, t 10, /
148          write new_state(i) as /, t 10, /
149      always
150          read number_signals(i) as i 10, /
151          write number_signals(i) as i 10, /
152          if number_signals(i) gt 0
153              reserve extevnt.origin(i,*), extevnt.destination(i,*),
154              extevnt.stype(i,*) as number_signals(i)
155              for j = 1 to number_signals(i)
156              do
157                  read extevnt.origin(i,j),
158                      extevnt.destination(i,j),
159                      extevnt.stype(i,j)
160                      as 3 t 10, /
161                  write extevnt.origin(i,j),
162                      extevnt.destination(i,j),
163                      extevnt.stype(i,j)
164                      as 3 t 10, /
165      loop
```

```
166      read new_strength(i) as i 10, /
167      write new_strength(i) as i 10, /
168      always
169      loop
170      always
171
172 end ''input
```

```

1 routine passive given component
2 ''
3 ''    Develops output signals for a passive component (no explicit
4 ''    command signals or power source).
5 ''
6 ''
7 ''          input process --- [ ] --- output process
8 ''
9 ''
10 '' Condensed decision table:
11 ''
12 ''      Process   Initial   Final    Process
13 ''      Case     Input    State    State   Output
14 ''      ----     -----   -----   -----  -----
15 ''      1        -       failed   failed  no
16 ''      2        no      standby  standby no
17 ''      3        yes     standby  failed  no
18 ''                  operating  operating yes
19 ''      4        no      operating standby  no
20 ''      5        yes     operating operating yes
21 ''
22     define rule as a saved 2-dimensional text array
23     define component as a pointer variable
24     define number.process, total.process, output.strength,
25         ruletype, success, and j as integer variables
26     define later.case as a saved integer variable
27 ''
28     Enter decision table.
29 ''
30     if later.case eq .no
31         reserve rule as 5 by 2
32         let rule(1,1) = ""           let rule(1,2) = "failed"
33         let rule(2,1) = "no"         let rule(2,2) = "standby"
34         let rule(3,1) = "yes"        let rule(3,2) = "standby"
35         let rule(4,1) = "no"         let rule(4,2) = "operating"
36         let rule(5,1) = "yes"        let rule(5,2) = "operating"
37         let later.case = .yes
38     always
39 ''
40 '' Determine input signal status.
41 ''
42     for every signal in input.sset(component)
43     do
44         if signal.type(signal) eq "process"
45             add 1 to total.process
46             if strength(signal) eq .on
47                 add 1 to number.process
48             always
49         always
50     loop
51 ''
52 '' Develop test vector for comparison with rules. Assume that
53 '' a single process signal is sufficient (i.e., an OR gate).
54 ''
55     if number.process ge 1

```

```
56      let test(1) = "yes"
57  else
58      let test(1) = "no"
59  always
60  let test(2) = state(component)
61 ''
62 ''  Determine appropriate rule.
63 ''  for ruletype = 1 to 5
64  do
65      for j = 1 to 2
66      do
67          if rule(ruletype,j) ne "" and rule(ruletype,j) ne test(j)
68              go to 'next'
69          always
70      loop
71      go to 'found'
72  'next'
73  loop
74 ''
75 ''  Select rule.
76 ''  'found'
77  select case ruletype
78
79  case 1
80      let state(component) = "failed"
81      let output.strength = .no
82
83  case 2
84      let state(component) = "standby"
85      let output.strength = .no
86
87  case 3
88      call demand.test giving component yielding success
89      if success eq .no
90          let state(component) = "failed"
91          let output.strength = .no
92      else
93          let state(component) = "operating"
94          let output.strength = .yes
95      always
96
97  case 4
98      let state(component) = "standby"
99      let output.strength = .no
100
101  case 5
102      let state(component) = "operating"
103      let output.strength = .yes
104
105  default
106
107 ''
108 ''  Error messages can be put here if rule not matched.
109 ''
110 ''
```

```
111 endselect
112 ''      Update output signals.
113 ''      for every signal in output.sset(component)
114      let strength(signal) = output.strength
115
116
117
118 return
119
120 end ''passive
```

```
1 process repair.supervisor
2 ''
3 ''    This process can be modified in the future to determine
4 ''    when a failed component should begin the repair process.
5 ''    Time delays can be inserted (repair delays) and if repair
6 ''    resources are limited the number of components under
7 ''    repair at any given time can be controlled here.
8 ''
9 ''    Currently this routine will be called from the system.update
10 ''   routine every time a new failure is detected. This routine
11 ''   uses the repair.probability for the failed component to
12 ''   determine if the component is repairable or not. If the
13 ''   component is repairable a repair is then begun immediately.
14 ''   To determine what the current status of each component is
15 ''   the status variable can be checked. The status will be
16 ''   working, resetting, awaiting repair, under repair, or not
17 ''   repairable.
18 ''
19 ''    This portion is for defining a repair delay.
20 ''
21     define component as a pointer variable
22     define a, b, and x as real variables
23     let a = 1.0
24     let b = 100.0
25     let x = time.v
26     if run.type eq 1
27       wait b hours
28       let a = 0.0
29       go to 'good'
30     otherwise
31 ''      wait weibull.f(a,b,1) hours
32 ''      'good'
33 ''
34 ''      If it is desireable to use various repair delays on a frequent
35 ''      basis, the program could be modified to read in the repair
36 ''      delay distribution parameters. The above delay is a weibull
37 ''      distribution, but with the parameters chosen, it is actually
38 ''      an exponential distribution.
39 ''
40     for every component in system.cset
41       with failure.time(component) eq x
42     find the first case
43     if found
44       if status(component) = .awaiting.repair
45         if random.f(1) le repair.probability(component)
46           resume the component
47         else
48           let status(component) = .not.repairable
49           always
50           always
51           let failure.time(component) = -1.0
52     else
53       print 1 line thus
54       In repair supervisor routine the component to repair was not IDed.
55     always
```

```
56
57     return
58
59 end ''repair.supervisor
```

```
1 routine run.initialize
2 ''
3 ''    initialization of components, signals, and external events
4 ''
5     define i, j, k, and signal.count as integer variables
6     define x, y, and z as real variables
7 ''
8 ''    Component initialization.
9 ''
10    reserve cptr(*) as n.component.record
11    for i = 1 to n.component.record
12    do
13        activate a component called cptr(i) now
14        file cptr(i) in system.cset
15        let name(cptr(i)) = trim.f(component_name(i),0)
16        let component.type(cptr(i)) = trim.f(component_type(i),0)
17        let n.input.sset(cptr(i)) = number_inputs(i)
18        let n.output.sset(cptr(i)) = number_outputs(i)
19        let demand.failure.frequency(cptr(i)) =
20            demand_failure_frequency(i)
21        let run.failure.frequency(cptr(i)) = run_failure_frequency(i)
22        let repair.probability(cptr(i)) = repair_probability(i)
23        let repair.function.shape(cptr(i)) = repair_function_shape(i)
24        let repair.function.scale(cptr(i)) = repair_function_scale(i)
25
26        select case component.type(cptr(i))
27
28        case "active"
29            let response.function(cptr(i)) = 'active'
30
31        case "passive"
32            let response.function(cptr(i)) = 'passive'
33
34        case "valve"
35            let response.function(cptr(i)) = 'valve'
36
37        case "check_valve"
38            let response.function(cptr(i)) = 'check.valve'
39
40        case "switch", "breaker"
41            let response.function(cptr(i)) = 'switch'
42
43        default
44            let response.function(cptr(i)) = 'active'
45            print 1 line with name(cptr(i)) thus
46            In initialize routine response function not matched to *****
47
48        endselect
49
50    loop
51    add 5 to total.signal.count
52    reserve sptr(*) as total.signal.count
53 ''
54 ''    Initialize and file boundary condition signals.
55 ''
```

''TANK'

```
56    for j = 1 to n.component.record
57    do
58      for k = 1 to number_inputs(j)
59      do
60        if trim.f(input.name(j,k),0) eq "system"
61          add 1 to signal.count
62          create a signal called sptr(signal.count)
63          let signal.type(sptr(signal.count)) =
64            trim.f(input.signal.type(j,k),0)
65          let origin(sptr(signal.count)) = "system"
66          let destination(sptr(signal.count)) =
67            trim.f(component_name(j),0)
68          file sptr(signal.count) in input.sset(cptr(j))
69          file sptr(signal.count) in system.boundary.sset
70          file sptr(signal.count) in system.sset
71        always
72      loop
73    loop
74  ''
75  ''  Initialize and file component output signals.
76  ''
77  for j = 1 to n.component.record
78  do
79    for k = 1 to number_outputs(j)
80    do
81      add 1 to signal.count
82      create a signal called sptr(signal.count)
83      let signal.type(sptr(signal.count)) =
84        trim.f(output.signal.type(j,k),0)
85      let origin(sptr(signal.count)) = trim.f(component_name(j),0)
86      let destination(sptr(signal.count)) =
87        trim.f(output.name(j,k),0)
88      for every component in system.cset
89        with name(component) eq destination(sptr(signal.count))
90        find the first case
91        if found
92          file sptr(signal.count) in input.sset(component)
93        else
94          if destination(sptr(signal.count)) eq "system"
95            file sptr(signal.count) in system.success.sset
96          always
97          always
98          file sptr(signal.count) in output.sset(cptr(j))
99          file sptr(signal.count) in system.sset
100     loop
101   loop
102  ''
103  ''  Create and initialize external events, using
104  ''  permanent entity external.event.record.
105  ''
106  if n.external.event.record gt 0
107    reserve eptr(*) as n.external.event.record
108    for i = 1 to n.external.event.record
109    do
110      activate an external.event called eptr(i) now
```

```
111      let occurrence.time(eptr(i)) = occurrence_time(i)
112      add .001 to occurrence.time(eptr(i))
113      let new.state(eptr(i)) = trim.f(new_state(i),0)
114      for j = 1 to number_components(i)
115      do
116          for every component in system.cset
117              with name(component) eq trim.f(extevnt.component(i,j),0)
118              find the first case
119              if found
120                  file component in extevnt.cset(eptr(i))
121                  always
122                      loop
123      let new.strength(eptr(i)) = new_strength(i)
124      let number.signals(eptr(i)) = number_signals(i)
125      if number.signals(eptr(i)) eq 1
126          let signal.origin(eptr(i)) = trim.f(extevnt.origin(i,1),0)
127          let signal.destination(eptr(i)) =
128              trim.f(extevnt.destination(i,1),0)
129          let signal.typeee(eptr(i)) = trim.f(extevnt.sstype(i,1),0)
130          always
131          file eptr(i) in system.eset
132      loop
133      always
134
135      reserve test as 4
136      reserve signal.status(*) as dim.f(sptr(*))
137      reserve unavailability.dist(*) as ntrial
138      reserve aptr(*) as ntimes
139      if distribution.type eq 1
140          let x = simulation.time / (ntimes - 1)
141          let time_avail(1) = 0.
142          for i = 2 to ntimes
143          do
144              let time_avail(i) = (i - 1) * x
145          loop
146      always
147      if distribution.type eq 2
148          let y = log.10.f(simulation.time)
149          let x = y / (ntimes - 1)
150          let time_avail(1) = 0.
151          for i = 2 to ntimes
152          do
153              let z = (i - 1) * x
154              let time_avail(i) = 10 ** z
155          loop
156      always
157      for i = 1 to ntimes
158      do
159          activate an availability called aptr(i) now
160          let time.avail(aptr(i)) = time_avail(i)
161      loop
162
163      return
164
165 end ''run.initialize
```

```
1 routine run.output
2 ''
3 ''      This routine will print the output report at the end of the
4 ''      run.  It prints the time dependent unavailability data and the
5 ''      average unavailability distribution data.
6 ''
7     define x as a real variable
8
9     for i = 1 to ntimes
10    do
11        let x = time.avail.data(aptr(i))
12        let time.avail.data(aptr(i)) = x / ntrial
13        let x = 1 - time.avail.data(aptr(i))
14        let time.avail.data(aptr(i)) = x
15    loop
16
17 write as *.,/
18 print 6 lines with ntrial thus
19                      AFTER **** TRIALS
20
21                         THE TIME DEPENDENT UNAVAILABILITY IS AS FOLLOWS
22
23                         TIME          UNAVAILABILITY
24  -----
25 for i = 1 to ntimes
26 do
27     print 2 lines with time.avail(aptr(i))
28     and time.avail.data(aptr(i)) thus
29
30                     *****.**          *.*****
31     loop
32 ''
33 ''      Sort the average unavailability distribution data.
34 ''
35     define l, m, n, j, k, and im as integer variables
36     define xp as a real variable
37
38     let m = ntrial
39     'sort1'
40     let l = m
41     let m = div.f(l,2)
42     if m gt 0
43         let k = ntrial - m
44         for j = 1 to k
45         do
46             let n = j
47             'sort2'
48             let im = n + m
49             if unavailability.dist(n) gt unavailability.dist(im)
50                 let xp = unavailability.dist(n)
51                 let unavailability.dist(n) = unavailability.dist(im)
52                 let unavailability.dist(im) = xp
53                 let l = n
54                 let n = l - m
55                 if n gt 0
```

```
56          go to 'sort2'
57          otherwise
58          always
59          loop
60          if m gt 0
61          go to 'sort1'
62          otherwise
63          always
64
65          write as *.,/
66          print 6 lines with ntrial and simulation.time thus
67          AFTER **** TRIALS
68          AND
69          OVER A TIME PERIOD OF ***** HOURS
70          THE AVERAGE SYSTEM UNAVAILABILITY IS AS FOLLOWS
71  -----
72
73          define x1, x5, x25, x40, x50, x60, x75, x95, and x99
74          as integer variables
75          let x1 = div.f(ntrial,100)
76          let x = 5 * ntrial
77          let x5 = div.f(x,100)
78          let x = 25 * ntrial
79          let x25 = div.f(x,100)
80          let x = 40 * ntrial
81          let x40 = div.f(x,100)
82          let x50 = div.f(ntrial,2)
83          let x = 60 * ntrial
84          let x60 = div.f(x,100)
85          let x = 75 * ntrial
86          let x75 = div.f(x,100)
87          let x = 95 * ntrial
88          let x95 = div.f(x,100)
89          let x = 99 * ntrial
90          let x99 = div.f(x,100)
91          if x1 eq 0
92          let x1 = 1
93          always
94          if x5 eq 0
95          let x5 = 1
96          always
97          print 27 lines with minimum.unavailability, unavailability.dist(x1),
98          unavailability.dist(x5), unavailability.dist(x25),
99          unavailability.dist(x40), unavailability.dist(x50),
100         unavailability.dist(x60),unavailability.dist(x75),
101         unavailability.dist(x95), unavailability.dist(x99),
102         maximum.unavailability, average.unavailability,
103         and variance.unavailability thus
104
105          The minimum is:      *****
106
107          The 1st percentile is: *****
108
109          The 5th percentile is: *****
110
```

```
111      The 25th percentile is: *.*****
112
113      The 40th percentile is: *.*****
114
115      The 50th percentile is: *.*****
116
117      The 60th percentile is: *.*****
118
119      The 75th percentile is: *.*****
120
121      The 95th percentile is: *.*****
122
123      The 99th percentile is: *.*****
124
125      The maximum is: *.*****
126
127      The mean is: *.*****
128
129      The variance is: *.*****
130
131 ''
132 ''    Use this portion to print out all of the average system
133 ''    unavailability values, one for every trial. These are the
134 ''    values on which the above percentiles are based.
135 ''
136 ''    write as *.,/
137 ''    for i = 1 to ntrial
138 ''    do
139 ''        print 1 line with i and unavailability.dist(i) thus
140 ''        point **** is *.*****
141 ''    loop
142
143 end ''run.output
```

```
1 process schedule.avail.samples
2 ''
3 ''    This process will cause samples to be taken at the designated
4 ''    times during each trial to compute the time dependent
5 ''    availability of the system.
6 ''
7     define x as a real variable
8
9     wait .002 hours
10    resume the availability called aptr(1)
11    for i = 2 to ntimes
12    do
13        let x = time.avail(aptr(i)) - time.avail(aptr(i - 1))
14        wait x hours
15        resume the availability called aptr(i)
16    loop
17
18    return
19
20 end ''schedule.avail.samples
```

```
1 process schedule.external.events
2 ''
3 ''    Schedules external events.
4 ''
5     define i as an integer variable
6     define x as a real variable
7
8     if n.external.event.record gt 0
9        wait occurrence.time(eptr(1)) hours
10       resume the external.event called eptr(1)
11       for i = 2 to dim.f(eptr(*))
12       do
13          let x = occurrence.time(eptr(i)) - occurrence.time(eptr(i - 1))
14          wait x hours
15          resume the external.event called eptr(i)
16       loop
17     always
18
19   return
20
21 end ''schedule.external.events
```

```
1 process stop.scenario
2 ''
3 ''    This process will interrupt any external events or components
4 ''    still scheduled to occur later in time. It then resets all
5 ''    components so they can begin operation again in the next trial.
6 ''
7     call system.update
8
9     for every external.event in ev.s(i.external.event)
10        interrupt external.event
11
12    for every component in ev.s(i.component)
13    do
14        interrupt component
15        let time.a(component) = 0.0
16    loop
17
18    for every component in system.cset
19    do
20        let status(component) = .reset.run
21        resume component
22    loop
23
24    return
25
26 end ''stop.scenario
```

```

1 routine switch given component
2 ''
3 ''    Develops output signals for a switch or breaker
4 ''    using explicit command signals. Assumes that the component
5 ''    has one or more command signal inputs, power inputs, and
6 ''    process inputs:
7 ''
8 ''        input command ---|_____|--- output process
9 ''        input power   ---|_____|--- output process
10 ''       input process ---|_____|--- output process
11 ''
12 ''    Condensed decision table:
13 ''
14 ''    Command      Power      Process      Initial      Final      Process
15 ''Case   Input      Input      Input      State       State      Output
16 ''----  -----  -----  -----  -----  -----  -----
17 ''    1      -      -      -      failed_open  failed_open  no
18 ''    2      -      no     -      open        open        no
19 ''    3      open    -      -      open        open        no
20 ''    4      none    -      -      open        open        no
21 ''    5      close   yes    no     open        failed_open  no
22 ''              -      -      -      -          closed      no
23 ''    6      close   yes    yes   open        failed_open  no
24 ''              -      -      -      -          closed      yes
25 ''    7      -      -      no    failed_closed failed_closed no
26 ''    8      -      -      yes   failed_closed failed_closed yes
27 ''    9      -      no     no     closed      closed      no
28 ''    10     -      no     yes   closed      closed      yes
29 ''    11     open   yes    no     closed      failed_closed no
30 ''              -      -      -      -          open       no
31 ''    12     open   yes    yes   closed      failed_closed yes
32 ''              -      -      -      -          open       no
33 ''    13     none   -      no     closed      closed      no
34 ''    14     none   -      yes   closed      closed      yes
35 ''    15     close   -      no     closed      closed      no
36 ''    16     close   -      yes   closed      closed      yes
37 ''
38     define rule as a saved 2-dimensional text array
39     define component as a pointer variable
40     define index.command, total.command, number.power, total.power,
41         number.process, total.process, output.strength, ruletype,
42         success and j as integer variables
43     define later.case as a saved integer variable
44 ''
45 ''    Enter decision table.
46 ''
47     if later.case eq .no
48     reserve rule as 16 by 4
49     let rule(1,1) = ""           let rule(1,2) = ""
50     let rule(1,3) = ""           let rule(1,4) = "failed_open"
51     let rule(2,1) = ""           let rule(2,2) = "no"
52     let rule(2,3) = ""           let rule(2,4) = "open"
53     let rule(3,1) = "open"      let rule(3,2) = ""
54     let rule(3,3) = ""           let rule(3,4) = "open"
55     let rule(4,1) = "none"       let rule(4,2) = ""

```

```
56      let rule(4,3) = ""      let rule(4,4) = "open"
57      let rule(5,1) = "close"  let rule(5,2) = "yes"
58      let rule(5,3) = "no"    let rule(5,4) = "open"
59      let rule(6,1) = "close"  let rule(6,2) = "yes"
60      let rule(6,3) = "yes"   let rule(6,4) = "open"
61      let rule(7,1) = ""      let rule(7,2) = ""
62      let rule(7,3) = "no"    let rule(7,4) = "failed_closed"
63      let rule(8,1) = ""      let rule(8,2) = ""
64      let rule(8,3) = "yes"   let rule(8,4) = "failed_closed"
65      let rule(9,1) = ""      let rule(9,2) = "no"
66      let rule(9,3) = "no"    let rule(9,4) = "closed"
67      let rule(10,1) = ""     let rule(10,2) = "no"
68      let rule(10,3) = "yes"  let rule(10,4) = "closed"
69      let rule(11,1) = "open"  let rule(11,2) = "yes"
70      let rule(11,3) = "no"   let rule(11,4) = "closed"
71      let rule(12,1) = "open"  let rule(12,2) = "yes"
72      let rule(12,3) = "yes"  let rule(12,4) = "closed"
73      let rule(13,1) = "none"  let rule(13,2) = ""
74      let rule(13,3) = "no"   let rule(13,4) = "closed"
75      let rule(14,1) = "none"  let rule(14,2) = ""
76      let rule(14,3) = "yes"  let rule(14,4) = "closed"
77      let rule(15,1) = "close"  let rule(15,2) = ""
78      let rule(15,3) = "no"   let rule(15,4) = "closed"
79      let rule(16,1) = "close"  let rule(16,2) = ""
80      let rule(16,3) = "yes"   let rule(16,4) = "closed"
81      let later.case = .yes
82  always
83  //
84  // Determine input signal status. Assume that "open" and "close"
85  // commands cancel each other out (respective values of 1 and -1).
86  //
87  for every signal in input.sset(component)
88  do
89      if signal.type(signal) eq "process"
90          add 1 to total.process
91          if strength(signal) eq .on
92              add 1 to number.process
93          always
94      else
95          if signal.type(signal) eq "power"
96              add 1 to total.power
97              if strength(signal) eq .on
98                  add 1 to number.power
99          always
100     else
101         add 1 to total.command
102         add strength(signal) to index.command
103     always
104  always
105  loop
106 //
107 // Develop test vector for comparison with rules. Assume that
108 // a single process signal is sufficient, and that a single power
109 // signal is sufficient (i.e., OR gates).
110 //
```

```
111      if index.command eq -1
112          let test(1) = "close"
113      else
114          if index.command eq 0
115              let test(1) = "none"
116          else
117              let test(1) = "open"
118          always
119      always
120      if number.power ge 1
121          let test(2) = "yes"
122      else
123          let test(2) = "no"
124      always
125      if number.process ge 1
126          let test(3) = "yes"
127      else
128          let test(3) = "no"
129      always
130      let test(4) = state(component)
131 ''
132 ''    Determine appropriate rule.
133 ''
134 for ruletype = 1 to 16
135 do
136     for j = 1 to 4
137         do
138             if rule(ruletype,j) ne "" and rule(ruletype,j) ne test(j)
139                 go to 'next'
140                 always
141                 loop
142                 go to 'found'
143             'next'
144             loop
145 ''
146 ''    Select rule.
147 ''
148     'found'
149     select case ruletype
150
151     case 1
152         let state(component) = "failed_open"
153         let output.strength = .no
154
155     case 2, 3, 4
156         let state(component) = "open"
157         let output.strength = .no
158
159     case 5
160         call demand.test giving component yielding success
161         if success eq .no
162             let state(component) = "failed_open"
163             let output.strength = .no
164         else
165             let state(component) = "closed"
```

```
166      let output.strength = .no
167      always
168
169      case 6
170          call demand.test giving component yielding success
171          if success eq .no
172              let state(component) = "failed_open"
173              let output.strength = .no
174          else
175              let state(component) = "closed"
176              let output.strength = .yes
177          always
178
179      case 7
180          let state(component) = "failed_closed"
181          let output.strength = .no
182
183      case 8
184          let state(component) = "failed_closed"
185          let output.strength = .yes
186
187      case 9, 13, 15
188          let state(component) = "closed"
189          let output.strength = .no
190
191      case 10, 14, 16
192          let state(component) = "closed"
193          let output.strength = .yes
194
195      case 11
196          call demand.test giving component yielding success
197          if success eq .no
198              let state(component) = "failed_closed"
199              let output.strength = .no
200          else
201              let state(component) = "open"
202              let output.strength = .no
203          always
204
205      case 12
206          call demand.test giving component yielding success
207          if success eq .no
208              let state(component) = "failed_closed"
209              let output.strength = .yes
210          else
211              let state(component) = "open"
212              let output.strength = .no
213          always
214
215      default
216 ''
217 ''    Error messages can be put here if rule not matched.
218 ''
219     endselect
220 ''
```

```
221 ''  Update output signals.  
222 ''  
223   for every signal in output.sset(component)  
224     let strength(signal) = output.strength  
225  
226   return  
227  
228 end ''switch
```

```
1 routine system.update
2 ''
3 ''    Updates status of signals in system, given status of all components
4 ''    Performs iterations until signals stabilize or number of iterations
5 ''    is exceeded.
6 ''
7 ''    Notes:
8 ''    1) Currently, maximum is set by number of signals.  Later
9 ''        versions might make use of digraph/Petri net results.
10 ''   2) Current version re-analyzes every component.  Later versions
11 ''      might only re-analyze components whose input changes.
12 ''
13     define rf as a subprogram variable
14     define i, itr, max.itr and number.success
15         as integer variables
16
17     for i = 1 to dim.f(sptr(*))
18         let signal.status(i) = strength(sptr(i))
19
20     let max.itr = dim.f(sptr(*))
21     for itr = 1 to max.itr
22         do
23 ''
24 ''    1) Check for changed component states and changed input
25 ''        signals.
26 ''    2) If found, place a demand on the component, and determine
27 ''        component response. (Later versions may activate signals
28 ''        here). Note that since output signals are updated
29 ''        in routine response.function, input signals for
30 ''        downstream components are also updated.
31 ''
32     for every component in system.cset
33     do
34         if state(component) ne old.state(component)
35             let rf = response.function(component)
36             call rf giving component
37         always
38         for every signal in input.sset(component)
39             with strength(signal) ne old.strength(signal)
40                 find the first case
41                 if found
42                     let rf = response.function(component)
43                     call rf giving component
44                 always
45             loop
46 ''
47 ''    Quit iteration if no changes to entire set of signals.
48 ''
49     for i = 1 to dim.f(sptr(*))
50         with strength(sptr(i)) ne signal.status(i)
51             find the first case
52             if found
53                 for i = 1 to dim.f(sptr(*))
54                     let signal.status(i) = strength(sptr(i))
55             else
```

```
56      go to 'update'
57      always
58  loop
59  print 2 lines with 24*time.v thus
60  !!! Error: Iteration maximum exceeded in routine system.update
61      time = ****.*** hours.
62 ''
63 ''  Activate newly started components, interrupt newly stopped
64 ''  components.
65 ''
66 'update'
67 for every component in system.cset
68 do
69   if status(component) eq .working
70     if state(component) ne old.state(component)
71       select case component.type(component)
72
73   case "active", "passive"
74     if state(component) eq "failed"
75       or state(component) eq "standby*"
76       or state(component) eq "operating*"
77     if old.state(component) eq "operating"
78       interrupt the component
79     always
80     let time.a(component) = 0.0
81     resume the component
82   always
83   if state(component) eq "standby"
84     and old.state(component) eq "operating"
85     interrupt the component
86   always
87   if state(component) eq "operating"
88     and old.state(component) eq "standby"
89     let time.a(component) = 0.0
90     let status(component) = .resetting
91     resume the component
92   always
93
94   case "check.valve", "switch", "valve"
95     if state(component) eq "closed"
96       and old.state(component) eq "open"
97       let status(component) = .resetting
98       interrupt the component
99       let time.a(component) = 0.0
100      resume the component
101    always
102    if state(component) eq "open"
103      and old.state(component) eq "closed"
104      let status(component) = .resetting
105      interrupt the component
106      let time.a(component) = 0.0
107      resume the component
108    always
109    if state(component) eq "failed_open"
110      or state(component) eq "failed_closed"
```

```
111         interrupt the component
112         let time.a(component) = 0.0
113         resume the component
114         always
115
116         default
117         print 1 line thus
118         When performing the system.update, no matching case!
119
120         endselect
121         always
122         always
123     loop
124 ''
125     '' Update status of system, components and signals.
126 ''
127     for every signal in system.success.sset
128     do
129         if strength(signal) eq .on
130             add 1 to number.success
131         always
132     loop
133     if number.success ge system.success.criterion
134         let system.state = "good"
135         let system.ind.var = 1
136     else
137         let system.state = "failed"
138         let system.ind.var = 0
139     always
140
141     call flow.update giving tptr(1)                      ''TANK
142
143     for every component in system.cset
144         let old.state(component) = state(component)
145
146     for every signal in system.sset
147         let old.strength(signal) = strength(signal)
148
149     return
150
151 end ''system.update
```

```
1 routine trial.initialize
2 ''
3 ''      This routine initializes the state of each component
4 ''      and the strength of each signal at the beginning of
5 ''      a trial.
6 ''
7     define i, j, and k as integer variables
8
9     let system.state = trim.f(initial.system.state,0)
10    if system.state eq "operating"
11        let system.ind.var = 1
12    else
13        let system.ind.var = 0
14    always
15 ''
16 ''      Component state initialization.
17 ''
18    for i = 1 to n.component.record
19    do
20        let old.state(cptr(i)) = trim.f(initial_state(i),0)
21        let state(cptr(i)) = old.state(cptr(i))
22    loop
23 ''
24 ''      Signal strength initialization.
25 ''
26    for i = 1 to n.component.record
27    do
28        for j = 1 to number_inputs(i)
29        do
30            for every signal in system.sset
31                with origin(signal) eq "system"
32                and destination(signal) eq trim.f(component_name(i),0)
33                and signal.type(signal) eq trim.f(input.signal.type(i,j),0)
34            find the first case
35            if found
36                let strength(signal) = input.signal.strength(i,j)
37            always
38        loop
39        for k = 1 to number_outputs(i)
40        do
41            for every signal in system.sset
42                with origin(signal) eq trim.f(component_name(i),0)
43                and destination(signal) eq trim.f(output.name(i,k),0)
44                and signal.type(signal) eq trim.f(output.signal.type(i,k),0)
45            find the first case
46            if found
47                let strength(signal) = output.signal.strength(i,k)
48            always
49        loop
50    loop
51
52    return
53
54 end ''trial.initialize
```

```

1 routine valve given component
2 ''
3 '' Develops output signals for an MOV or manual valve
4 '' using explicit command signals. Assumes that the component
5 '' has one or more command signal inputs, power inputs, and
6 '' process inputs:
7 ''
8 ''      input command --- | [ ] |
9 ''      input power   --- | [ ] | --- output process
10 ''     input process --- | [ ] |
11 ''
12 '' Condensed decision table:
13 ''
14 ''      Command    Power   Process   Initial      Final       Process
15 '' Case   Input     Input    Input     State        State      Output
16 '' -----  -----  -----  -----  -----
17 ''     1      -      -      -      failed_closed  failed_closed  no
18 ''     2      -      no     -      closed        closed      no
19 ''     3      close   -      -      closed        closed      no
20 ''     4      none   -      -      closed        closed      no
21 ''     5      open   yes    no     closed        failed_closed  no
22 ''           open   yes    yes    closed        failed_closed  no
23 ''           open   yes    yes    closed        failed_closed  no
24 ''           open   yes    yes    open         open        yes
25 ''     7      -      -      no     failed_open   failed_open   no
26 ''     8      -      -      yes    failed_open   failed_open   yes
27 ''     9      -      no     no     open         open        no
28 ''     10     -      no     yes    open         open        yes
29 ''     11     close  yes    no     open         failed_open  no
30 ''           close  yes    yes    open         failed_open  no
31 ''           close  yes    yes    open         failed_open  yes
32 ''           close  yes    yes    open         closed      no
33 ''     13     none   -      no     open         open        no
34 ''     14     none   -      yes    open         open        yes
35 ''     15     open   -      no     open         open        no
36 ''     16     open   -      yes    open         open        yes
37 ''
38 define rule as a saved 2-dimensional text array
39 define component as a pointer variable
40 define index.command, total.command, number.power, total.power,
41   number.process, total.process, output.strength, ruletype,
42   success and j as integer variables
43 define later.case as a saved integer variable
44 ''
45 '' Enter decision table.
46 ''
47 if later.case eq .no
48   reserve rule as 16 by 4
49   let rule(1,1) = ""      let rule(1,2) = ""
50   let rule(1,3) = ""      let rule(1,4) = "failed_closed"
51   let rule(2,1) = ""      let rule(2,2) = "no"
52   let rule(2,3) = ""      let rule(2,4) = "closed"
53   let rule(3,1) = "close" let rule(3,2) = ""
54   let rule(3,3) = ""      let rule(3,4) = "closed"
55   let rule(4,1) = "none"  let rule(4,2) = ""

```

```
56      let rule(4,3) = ""          let rule(4,4) = "closed"
57      let rule(5,1) = "open"       let rule(5,2) = "yes"
58      let rule(5,3) = "no"         let rule(5,4) = "closed"
59      let rule(6,1) = "open"       let rule(6,2) = "yes"
60      let rule(6,3) = "yes"        let rule(6,4) = "closed"
61      let rule(7,1) = ""          let rule(7,2) = ""
62      let rule(7,3) = "no"         let rule(7,4) = "failed_open"
63      let rule(8,1) = ""          let rule(8,2) = ""
64      let rule(8,3) = "yes"        let rule(8,4) = "failed_open"
65      let rule(9,1) = ""          let rule(9,2) = "no"
66      let rule(9,3) = "no"         let rule(9,4) = "open"
67      let rule(10,1) = ""         let rule(10,2) = "no"
68      let rule(10,3) = "yes"       let rule(10,4) = "open"
69      let rule(11,1) = "close"     let rule(11,2) = "yes"
70      let rule(11,3) = "no"         let rule(11,4) = "open"
71      let rule(12,1) = "close"     let rule(12,2) = "yes"
72      let rule(12,3) = "yes"       let rule(12,4) = "open"
73      let rule(13,1) = "none"      let rule(13,2) = ""
74      let rule(13,3) = "no"         let rule(13,4) = "open"
75      let rule(14,1) = "none"      let rule(14,2) = ""
76      let rule(14,3) = "yes"        let rule(14,4) = "open"
77      let rule(15,1) = "open"       let rule(15,2) = ""
78      let rule(15,3) = "no"         let rule(15,4) = "open"
79      let rule(16,1) = "open"       let rule(16,2) = ""
80      let rule(16,3) = "yes"        let rule(16,4) = "open"
81      let later.case = .yes
82 always
83 ''
84 '' Determine input signal status. Assume that "open" and "close"
85 '' commands cancel each other out (respective values of 1 and -1).
86 ''
87 for every signal in input.sset(component)
88 do
89     if signal.type(signal) eq "process"
90         add 1 to total.process
91         if strength(signal) eq .on
92             add 1 to number.process
93         always
94     else
95         if signal.type(signal) eq "power"
96             add 1 to total.power
97             if strength(signal) eq .on
98                 add 1 to number.power
99             always
100        else
101            add 1 to total.command
102            add strength(signal) to index.command
103        always
104    always
105 loop
106 ''
107 '' Develop test vector for comparison with rules. Assume that
108 '' a single process signal is sufficient, and that a single power
109 '' signal is sufficient (i.e., OR gates).
110 ''
```

```
111  if index.command eq -1
112    let test(1) = "close"
113  else
114    if index.command eq 0
115      let test(1) = "none"
116    else
117      let test(1) = "open"
118    always
119  always
120  if number.power ge 1
121    let test(2) = "yes"
122  else
123    let test(2) = "no"
124  always
125  "
126  " By changing the test for number of process inputs, it is
127  " possible to simulate k-out-of-n components.
128  "
129  if number.process ge 1
130    let test(3) = "yes"
131  else
132    let test(3) = "no"
133  always
134  let test(4) = state(component)
135  "
136  " Determine appropriate rule.
137  "
138  for ruletype = 1 to 16
139  do
140    for j = 1 to 4
141    do
142      if rule(ruletype,j) ne "" and rule(ruletype,j) ne test(j)
143        go to 'next'
144        always
145        loop
146        go to 'found'
147  'next'
148  loop
149  "
150  " Select rule.
151  "
152  'found'
153  select case ruletype
154
155  case 1
156    let state(component) = "failed_closed"
157    let output.strength = .no
158
159  case 2, 3, 4
160    let state(component) = "closed"
161    let output.strength = .no
162
163  case 5
164    call demand.test giving component yielding success
165    if success eq .no
```

```
166      let state(component) = "failed_closed"
167      let output.strength = .no
168  else
169      let state(component) = "open"
170      let output.strength = .no
171  always
172
173 case 6
174  call demand.test giving component yielding success
175  if success eq .no
176      let state(component) = "failed_closed"
177      let output.strength = .no
178  else
179      let state(component) = "open"
180      let output.strength = .yes
181  always
182
183 case 7
184  let state(component) = "failed_open"
185  let output.strength = .no
186
187 case 8
188  let state(component) = "failed_open"
189  let output.strength = .yes
190
191 case 9, 13, 15
192  let state(component) = "open"
193  let output.strength = .no
194
195 case 10, 14, 16
196  let state(component) = "open"
197  let output.strength = .yes
198
199 case 11
200  call demand.test giving component yielding success
201  if success eq .no
202      let state(component) = "failed_open"
203      let output.strength = .no
204  else
205      let state(component) = "closed"
206      let output.strength = .no
207  always
208
209 case 12
210  call demand.test giving component yielding success
211  if success eq .no
212      let state(component) = "failed_open"
213      let output.strength = .yes
214  else
215      let state(component) = "closed"
216      let output.strength = .no
217  always
218
219 , default
220 ,,
```

```
221 ''  Error messages can be put here if rule not matched.  
222 ''  
223   endselect  
224 ''  Update output signals.  
225 ''  
226 ''  
227   for every signal in output.sset(component)  
228     let strength(signal) = output.strength  
229  
230   return  
231  
232 end ''valve
```

Appendix C
TANK Program Listing

```
1 routine flow.update given tank
2 ''
3 ''      Determine the new flow rate if it has changed.
4 ''
5     define tank as a pointer variable
6     let flow.rate.in(tank) = 0
7     let flow.rate.out(tank) = 0
8     for every component in tank.input.cset(tank)
9     do
10        if name(component) eq "unit2"
11            if state(component) eq "open"
12                or state(component) eq "failed_open"
13                add 0.01 to flow.rate.in(tank)
14        always
15    else
16        if name(component) eq "unit3"
17            if state(component) eq "open"
18                or state(component) eq "failed_open"
19                add 0.005 to flow.rate.in(tank)
20        always
21    always
22    always
23 loop
24 for every component in tank.output.cset(tank)
25 do
26    if state(component) eq "open"
27        or state(component) eq "failed_open"
28        add 0.01 to flow.rate.out(tank)
29    always
30 loop
31
32 return
33
34 end ''flow.update
```

```
1 process stop.tank
2 ''
3 ''    This process will reset the tank process so it is ready
4 ''    for the execution of another trial.
5 ''
6     for every tank in ev.s(i.tank)
7     do
8         interrupt the tank
9     loop
10
11    for every tank in system.tset
12    do
13        let level(tank) = 100.0
14        let time.a(tank) = 0.0
15        resume the tank
16    loop
17
18    return
19
20 end ''stop.tank
```

```
1 process tank
2 ''
3 ''    This routine will continuously monitor the water level
4 ''    in a tank.
5 ''
6   'tankreset'
7   suspend
8   while time.v lt (simulation.time + 10)
9   do
10  ''
11  ''    This portion of the routine determines if the tank is in the
12  ''    proper control region and calls the tank update routine to
13  ''    make changes if necessary.
14  ''
15  work continuusly evaluating 'water.level' testing 'tank.condition'
16  let net.flow.rate(tank) = flow.rate.in(tank) - flow.rate.out(tank)
17  if level(tank) gt 90.0
18  go to 'tankreset'
19  otherwise
20  call tank.update giving tank
21  if level(tank) gt high.level(tank)
22  or level(tank) lt low.level(tank)
23  suspend
24  go to 'tankreset'
25  always
26
27  loop
28
29  suspend
30
31 end ''tank
```

```
1 function tank.condition(tank)
2 ''
3 ''      This function will cause calling of the tank update
4 ''      routine if the tank status is not satisfactory.
5 ''
6     define tank as a pointer variable
7 ''
8 ''      Use this method to adjust tank flow rate only at the
9 ''      end of integration time steps.
10 ''
11    define x as a real variable
12    let x = flow.rate.in(tank) - flow.rate.out(tank)
13    if net.flow.rate(tank) ne x
14        return with 1
15    otherwise
16 ''
17 ''      Is the tank too full?
18 ''
19    if level(tank) gt high.level(tank)
20        return with 1
21    otherwise
22 ''
23 ''      Is the tank too empty?
24 ''
25    if level(tank) lt low.level(tank)
26        return with 1
27    otherwise
28 ''
29 ''      Is the tank level high and the control state wrong?
30 ''
31    if level(tank) gt high.set(tank)
32        for every component in system.cset
33        do
34            if name(component) eq "unit1"
35                and state(component) eq "closed"
36                    return with 1
37            otherwise
38            if name(component) eq "unit2"
39                and state(component) eq "open"
40                    return with 1
41            otherwise
42            if name(component) eq "unit3"
43                and state(component) eq "open"
44                    return with 1
45            otherwise
46        loop
47    always
48 ''
49 ''      Is the tank level low and the control state wrong?
50 ''
51    if level(tank) lt low.set(tank)
52        for every component in system.cset
53        do
54            if name(component) eq "unit1"
55                and state(component) eq "open"
```

```
56         return with 1
57     otherwise
58     if name(component) eq "unit2"
59         and state(component) eq "closed"
60         return with 1
61     otherwise
62     if name(component) eq "unit3"
63         and state(component) eq "closed"
64         return with 1
65     otherwise
66     loop
67 always
68 ''
69 ''    Is the tank level satisfactory and the control state wrong?
70 ''
71     if level(tank) le high.set(tank)
72         and level(tank) ge low.set(tank)
73     for every component in system.cset
74     do
75         if name(component) eq "unit1"
76             and state(component) eq "closed"
77             return with 1
78         otherwise
79         if name(component) eq "unit2"
80             and state(component) eq "closed"
81             return with 1
82         otherwise
83         if name(component) eq "unit3"
84             and state(component) eq "open"
85             return with 1
86         otherwise
87     loop
88 always
89         return with 0
90
91 end ''tank.condition
```

```
1 routine tank.initialize.run
2 ''
3 ''      This routine initializes all of the variables associated
4 ''      with the Aldemir Tank Problem.  Initializes for the number
5 ''      of trials to be performed.
6 ''
7 define signal.count as an integer variable
8 let integrator.v = 'runge.kutta.r'
9 let max.step.v = 0.04166666666667      '' Approximately 1 hour
10 let min.step.v = 0.04166666666667     '' Approximately 1 hour
11 let abs.err.v = 0.001
12 let rel.err.v = 0.1
13 ''
14 ''      Create a tank.
15 ''
16 reserve tptr(*) as l
17 activate a tank called tptr(1) now
18 file tptr(1) in system.tset
19 let high.level(tptr(1)) = 3.0
20 let low.level(tptr(1)) = -3.0
21 let high.set(tptr(1)) = 1.0
22 let low.set(tptr(1)) = -1.0
23 ''
24 ''      Must create all of the Tank output signals since the base
25 ''      program does not recognize the tank as a component.  These
26 ''      signals include three command signals (one to each valve),
27 ''      the tank process output to the outlet valve, and the process
28 ''      output signal to the system for system status checking.
29 ''
30 let signal.count = 9
31 create a signal called sptr(signal.count)
32 let signal.type(sptr(signal.count)) = "command"
33 let origin(sptr(signal.count)) = "tank"
34 let destination(sptr(signal.count)) = "unit1"
35     for every component in system.cset
36         with name(component) eq "unit1"
37         find the first case
38         if found
39             file sptr(signal.count) in input.sset(component)
40             always
41             file sptr(signal.count) in tank.output.sset(tptr(1))
42             file sptr(signal.count) in system.sset
43 ''
44 add 1 to signal.count
45 create a signal called sptr(signal.count)
46 let signal.type(sptr(signal.count)) = "command"
47 let origin(sptr(signal.count)) = "tank"
48 let destination(sptr(signal.count)) = "unit2"
49     for every component in system.cset
50         with name(component) eq "unit2"
51         find the first case
52         if found
53             file sptr(signal.count) in input.sset(component)
54             always
55             file sptr(signal.count) in tank.output.sset(tptr(1))
```

```
56     file sptr(signal.count) in system.sset
57  ,,
58     add 1 to signal.count
59     create a signal called sptr(signal.count)
60     let signal.type(sptr(signal.count)) = "command"
61     let origin(sptr(signal.count)) = "tank"
62     let destination(sptr(signal.count)) = "unit3"
63         for every component in system.cset
64             with name(component) eq "unit3"
65             find the first case
66             if found
67                 file sptr(signal.count) in input.sset(component)
68                 always
69                 file sptr(signal.count) in tank.output.sset(tptr(1))
70                 file sptr(signal.count) in system.sset
71  ,
72     add 1 to signal.count
73     create a signal called sptr(signal.count)
74     let signal.type(sptr(signal.count)) = "process"
75     let origin(sptr(signal.count)) = "tank"
76     let destination(sptr(signal.count)) = "unit1"
77         for every component in system.cset
78             with name(component) eq "unit1"
79             find the first case
80             if found
81                 file sptr(signal.count) in input.sset(component)
82                 always
83                 file sptr(signal.count) in tank.output.sset(tptr(1))
84                 file sptr(signal.count) in system.sset
85  ,
86     add 1 to signal.count
87     create a signal called sptr(signal.count)
88     let signal.type(sptr(signal.count)) = "process"
89     let origin(sptr(signal.count)) = "tank"
90     let destination(sptr(signal.count)) = "system"
91     file sptr(signal.count) in tank.output.sset(tptr(1))
92     file sptr(signal.count) in system.sset
93     file sptr(signal.count) in system.success.sset
94     for every component in system.cset
95     do
96         for every signal in output.sset(component)
97         do
98             if destination(signal) eq "tank"
99                 file signal in tank.input.sset(tptr(1))
100                file component in tank.input.cset(tptr(1))
101                always
102            loop
103            for every signal in input.sset(component)
104                with signal.type(signal) eq "process"
105                do
106                    if origin(signal) eq "tank"
107                        file component in tank.output.cset(tptr(1))
108                        always
109                    loop
110                loop
```

111
112 return
113
114 end ''tank.initialize.run

```
1 routine tank.initialize.trial
2 ''
3 ''      This routine will reset the appropriate values to begin
4 ''      a new trial with the tank operating correctly.
5 ''
6     let level(tptr(1)) = 0.0
7     let net.flow.rate(tptr(1)) = 0.0
8     for every signal in tank.output.sset(tptr(1))
9       do
10      ''
11      ''      Turn on the flow output and test signal from the tank.
12      ''
13      if signal.type(signal) = "process"
14        let strength(signal) = .on
15      always
16      ''
17      ''      Turn off the command signals for the valves to change position.
18      ''
19      if signal.type(signal) = "command"
20        let strength(signal) = .off
21      always
22    loop
23
24    return
25
26 end ''tank.initialize.trial
```

```
1 routine tank.update given tank
2 ''
3 ''      This routine determines the flow going in and out of the
4 ''      tank and controls the opening and closing of the inlet and
5 ''      outlet valves. If the tank should happen to dryout or over
6 ''      flow this routine will suspend the tank routine.
7 ''
8     define tank as a pointer variable
9 ''
10 ''     This is to track dryout.
11 ''
12     if level(tank) lt low.level(tank)
13         for every signal in tank.output.sset(tank)
14             with signal.type(signal) eq "process"
15             do
16                 let strength(signal) = .no
17             loop
18             go to 'leave'
19     otherwise
20 ''
21 ''     This is to track overflow.
22 ''
23     if level(tank) gt high.level(tank)
24         for every signal in tank.output.sset(tank)
25             with destination(signal) eq "system"
26             do
27                 let strength(signal) = .no
28             loop
29             go to 'leave'
30     otherwise
31     if level(tank) lt low.set(tank)
32 ''
33 ''     Close the outlet valve and open both inlet valves.
34 ''
35     for every component in tank.output.cset(tank)
36     do
37         for every signal in input.sset(component)
38             with signal.type(signal) eq "command"
39             do
40                 let strength(signal) = -1
41             loop
42     loop
43     for every component in tank.input.cset(tank)
44     do
45         for every signal in input.sset(component)
46             with signal.type(signal) eq "command"
47             do
48                 let strength(signal) = 1
49             loop
50     loop
51     go to 'leave'
52     otherwise
53     if level(tank) gt high.set(tank)
54 ''
55 ''     Open the outlet valve and close both inlet valves.
```

```
56  ''          for every component in tank.output.cset(tank)
57  do           for every signal in input.sset(component)
58    with signal.type(signal) eq "command"
59    do           let strength(signal) = 1
60    loop
61  loop
62  for every component in tank.input.cset(tank)
63  do           for every signal in input.sset(component)
64    with signal.type(signal) eq "command"
65    do           let strength(signal) = -1
66    loop
67  loop
68  go to 'leave'
69 otherwise
70 ''
71  If the level of the tank is in the operating range,
72  open the outlet valve(unit1) and the inlet valve from
73  unit2, but close the inlet valve from unit3.
74 ''
75  for every component in tank.output.cset(tank)
76  do           for every signal in input.sset(component)
77    with signal.type(signal) eq "command"
78    do           let strength(signal) = 1
79    loop
80  loop
81  for every component in tank.input.cset(tank)
82  do           if name(component) eq "unit2"
83    for every signal in input.sset(component)
84      with signal.type(signal) eq "command"
85    do           let strength(signal) = 1
86    loop
87  loop
88  for every component in tank.input.cset(tank)
89  do           if name(component) eq "unit3"
90    for every signal in input.sset(component)
91      with signal.type(signal) eq "command"
92    do           let strength(signal) = -1
93    loop
94  loop
95 else
96   if name(component) eq "unit3"
97     for every signal in input.sset(component)
98       with signal.type(signal) eq "command"
99     do           let strength(signal) = -1
100    loop
101   always
102   always
103  loop
104  'leave'
105 call system.update
106 return
107
108
109
110 end ''tank.update
```

```
1 routine water.level(tank)
2 ''
3 ''    This routine supplies the integration rule for the continuous
4 ''    variable level of the tank.
5 ''
6     define tank as a pointer variable
7         let d.level(tank) = net.flow.rate(tank)*1440.0
8 ''
9 ''    We have left the time step as days and are reading flow rates
10 ''   as meter level change per minute thus the factor of 1440 above.
11 ''
12 end ''water.level
```

Appendix D
Sample Input Files

SINGLE COMPONENT, EXP REPAIR AND FAILURE, DUAL REPAIR STATES
10000.00 Time of simulation
0 Type of run (0 for normal)
100 Number of trials
21 Number of time points
1 Type of time distribution
1 Number of components
COMPONENT passive operating 1 1 Component one
0.0 0.01 Failure data
1.0 100.0 1.0 Repair data
system process 1 Input signal
system process 1 Output signal
standby 1 Initial system state
0 System success criteria
Number of external events

TWO OUT OF THREE PUMPS, EXPONENTIAL FAILURE AND REPAIR.

| | | | | | |
|----------|---------|-----------|---|---|----------------------------|
| 10000.00 | | | | | Time of simulation |
| 0 | | | | | Type of run (0 for normal) |
| 100 | | | | | Number of trials |
| 21 | | | | | Number of time points |
| 1 | | | | | Type of time distribution |
| 4 | | | | | Number of components |
| PUMP1 | active | operating | 3 | 1 | Component one |
| 0.0 | 0.01 | | | | Failure data |
| 1.0 | 100.0 | 1.0 | | | Repair data |
| system | power | | 1 | | Input signal |
| system | command | | 1 | | Input signal |
| system | process | | 1 | | Input signal |
| VALVE | process | | 1 | | Output signal |
| PUMP2 | active | operating | 3 | 1 | Component two |
| 0.0 | 0.01 | | | | Failure data |
| 1.0 | 100.0 | 1.0 | | | Repair data |
| system | power | | 1 | | Input signal |
| system | command | | 1 | | Input signal |
| system | process | | 1 | | Input signal |
| VALVE | process | | 1 | | Output signal |
| PUMP3 | active | operating | 3 | 1 | Component three |
| 0.0 | 0.01 | | | | Failure data |
| 1.0 | 100.0 | 1.0 | | | Repair data |
| system | power | | 1 | | Input signal |
| system | command | | 1 | | Input signal |
| system | process | | 1 | | Input signal |
| VALVE | process | | 1 | | Output signal |
| VALVE | valve | open | 5 | 1 | Component four |
| 0.0 | 0.01 | | | | Failure data |
| 1.0 | 100.0 | 1.0 | | | Repair data |
| system | power | | 1 | | Input signal |
| system | command | | 1 | | Input signal |
| PUMP1 | process | | 1 | | Input signal |
| PUMP2 | process | | 1 | | Input signal |
| PUMP3 | process | | 1 | | Input signal |
| system | process | | 1 | | Output signal |
| standby | | | | | Initial system state |
| | 1 | | | | System success criteria |
| | 0 | | | | Number of external events |

SIMULATION OF GO-FLOW LIGHT BULB PROBLEM

| | | | | |
|---------|---------|---------|---|----------------------------------|
| 20.00 | | | | Time of simulation |
| 0 | | | | Type of run (0 for normal) |
| 1000 | | | | Number of trials |
| 7 | | | | Number of time points |
| 0 | | | | Type of time distribution |
| 0.00 | | | | Time points |
| 1.00 | | | | |
| 9.99 | | | | |
| 10.00 | | | | |
| 11.00 | | | | |
| 15.00 | | | | |
| 20.00 | | | | |
| 5 | | | | Number of components |
| BATTERY | passive | standby | 1 | Component number one |
| 0.1 | 0.0 | | | Failure data |
| 1.0 | 1.0 | 0.0 | | Repair data |
| system | process | | 0 | Input signal |
| SWITCH1 | process | | 0 | Output signal |
| SWITCH2 | process | | 0 | Output signal |
| SWITCH1 | switch | open | 3 | Component number two |
| 0.3 | 0.0 | | | Failure data |
| 1.0 | 1.0 | 0.0 | | Repair data |
| system | command | | 0 | Input signal |
| system | power | | 1 | Input signal |
| BATTERY | process | | 0 | Input signal |
| LIGHT1 | process | | 0 | Output signal |
| SWITCH2 | switch | open | 3 | Component number three |
| 0.3 | 0.0 | | | Failure data |
| 1.0 | 1.0 | 0.0 | | Repair data |
| system | command | | 0 | Input signal |
| system | power | | 1 | Input signal |
| BATTERY | process | | 0 | Input signal |
| LIGHT2 | process | | 0 | Output signal |
| LIGHT1 | passive | standby | 1 | Component number four |
| 0.2 | 0.001 | | | Failure data |
| 1.0 | 1.0 | 0.0 | | Repair data |
| SWITCH1 | process | | 0 | Input signal |
| system | process | | 0 | Output signal |
| LIGHT2 | passive | standby | 1 | Component number five |
| 0.2 | 0.001 | | | Failure data |
| 1.0 | 1.0 | 0.0 | | Repair data |
| SWITCH2 | process | | 0 | Input signal |
| system | process | | 0 | Output signal |
| standby | | | | Initial system state |
| 1 | | | | System success criteria |
| 3 | | | | Number of external events |
| 0.00 | | 0 | | External event #1, Time, #Comps. |
| 1 | | | | Number signals |
| system | BATTERY | process | | Signal |
| 1 | | | | New strength |
| 0.00 | | 0 | | External event #2, Time, #Comps. |
| 1 | | | | Number signals |
| system | SWITCH1 | command | | Signal |
| -1 | | | | New strength |
| 10.00 | | 0 | | External event #3, Time, #Comps. |
| 1 | | | | Number signals |
| system | SWITCH2 | command | | Signal |
| -1 | | | | New strength |

TEST OF THE TANK PORTION OF THE PROGRAM

| | | | | | |
|---------|---------|---------|-----|----|---|
| | 1000.00 | | | | |
| | 0 | | | | |
| | 1000 | | | | |
| | 201 | | | | |
| | 1 | | | | |
| | 3 | | | | |
| unit1 | valve | open | | 3 | 1 |
| | 0.0 | 0.00312 | | | |
| | 1.0 | 1.0 | 0.0 | | |
| | system | power | | 1 | |
| | tank | process | | 1 | |
| | tank | command | | 1 | |
| | nowhere | process | | 1 | |
| unit2 | valve | open | | 3 | 1 |
| | 0.0 | 0.00456 | | | |
| | 1.0 | 1.0 | 0.0 | | |
| | system | power | | 1 | |
| | system | process | | 1 | |
| | tank | command | | 1 | |
| | tank | process | | 1 | |
| unit3 | valve | closed | | 3 | 1 |
| | 0.0 | 0.0057 | | | |
| | 1.0 | 1.0 | 0.0 | | |
| | system | power | | 1 | |
| | system | process | | 1 | |
| | tank | command | | -1 | |
| | tank | process | | 0 | |
| standby | | | | | |
| | 1 | | | | |
| | 0 | | | | |

Appendix E
Sample Output Files

SINGLE COMPONENT, EXP REPAIR AND FAILURE, DUAL REPAIR STATES
10000.00
0
100
21
1
1
COMPONENT passive operating 1 1
0. .01000
1.00000 100.00000 1.00000
system process 1
system process 1
standby
1
0

AFTER 100 TRIALS
AND
OVER A TIME PERIOD OF 10000 HOURS
THE AVÉRAGE SYSTEM UNAVAILABILITY IS AS FOLLOWS

The minimum is: .5510
The 1st percentile is: .5510
The 5th percentile is: .5804
The 25th percentile is: .6343
The 40th percentile is: .6538
The 50th percentile is: .6618
The 60th percentile is: .6740
The 75th percentile is: .7002
The 95th percentile is: .7440
The 99th percentile is: .7579
The maximum is: .7732
The mean is: .6644
The variance is: .0023

AFTER 100 TRIALS

THE TIME DEPENDENT UNAVAILABILITY IS AS FOLLOWS

| TIME | UNAVAILABILITY |
|----------|----------------|
| 0. | 0. |
| 500.00 | .6300 |
| 1000.00 | .7000 |
| 1500.00 | .7000 |
| 2000.00 | .6800 |
| 2500.00 | .6700 |
| 3000.00 | .6500 |
| 3500.00 | .6700 |
| 4000.00 | .7200 |
| 4500.00 | .6900 |
| 5000.00 | .6400 |
| 5500.00 | .5900 |
| 6000.00 | .6300 |
| 6500.00 | .6800 |
| 7000.00 | .6500 |
| 7500.00 | .6800 |
| 8000.00 | .6900 |
| 8500.00 | .6800 |
| 9000.00 | .6100 |
| 9500.00 | .7000 |
| 10000.00 | .6400 |

| | | | |
|-------|----|----|-------|
| point | 1 | is | .5510 |
| point | 2 | is | .5622 |
| point | 3 | is | .5700 |
| point | 4 | is | .5787 |
| point | 5 | is | .5804 |
| point | 6 | is | .5836 |
| point | 7 | is | .5883 |
| point | 8 | is | .5956 |
| point | 9 | is | .5962 |
| point | 10 | is | .5967 |
| point | 11 | is | .5976 |
| point | 12 | is | .5997 |
| point | 13 | is | .6006 |
| point | 14 | is | .6056 |
| point | 15 | is | .6095 |
| point | 16 | is | .6121 |
| point | 17 | is | .6122 |
| point | 18 | is | .6167 |
| point | 19 | is | .6179 |
| point | 20 | is | .6223 |
| point | 21 | is | .6233 |
| point | 22 | is | .6264 |
| point | 23 | is | .6321 |
| point | 24 | is | .6342 |
| point | 25 | is | .6343 |
| point | 26 | is | .6371 |
| point | 27 | is | .6374 |
| point | 28 | is | .6399 |
| point | 29 | is | .6414 |
| point | 30 | is | .6430 |
| point | 31 | is | .6444 |
| point | 32 | is | .6454 |
| point | 33 | is | .6464 |
| point | 34 | is | .6465 |
| point | 35 | is | .6477 |
| point | 36 | is | .6481 |
| point | 37 | is | .6494 |
| point | 38 | is | .6500 |
| point | 39 | is | .6525 |
| point | 40 | is | .6538 |
| point | 41 | is | .6540 |
| point | 42 | is | .6544 |
| point | 43 | is | .6547 |
| point | 44 | is | .6555 |
| point | 45 | is | .6568 |
| point | 46 | is | .6586 |
| point | 47 | is | .6597 |
| point | 48 | is | .6617 |
| point | 49 | is | .6617 |
| point | 50 | is | .6618 |
| point | 51 | is | .6620 |
| point | 52 | is | .6626 |
| point | 53 | is | .6633 |

| | | | |
|-------|-----|----|-------|
| point | 54 | is | .6647 |
| point | 55 | is | .6661 |
| point | 56 | is | .6671 |
| point | 57 | is | .6674 |
| point | 58 | is | .6680 |
| point | 59 | is | .6694 |
| point | 60 | is | .6740 |
| point | 61 | is | .6742 |
| point | 62 | is | .6756 |
| point | 63 | is | .6763 |
| point | 64 | is | .6821 |
| point | 65 | is | .6835 |
| point | 66 | is | .6850 |
| point | 67 | is | .6875 |
| point | 68 | is | .6876 |
| point | 69 | is | .6879 |
| point | 70 | is | .6922 |
| point | 71 | is | .6932 |
| point | 72 | is | .6967 |
| point | 73 | is | .6978 |
| point | 74 | is | .6996 |
| point | 75 | is | .7002 |
| point | 76 | is | .7023 |
| point | 77 | is | .7040 |
| point | 78 | is | .7049 |
| point | 79 | is | .7064 |
| point | 80 | is | .7064 |
| point | 81 | is | .7084 |
| point | 82 | is | .7085 |
| point | 83 | is | .7097 |
| point | 84 | is | .7136 |
| point | 85 | is | .7146 |
| point | 86 | is | .7180 |
| point | 87 | is | .7185 |
| point | 88 | is | .7218 |
| point | 89 | is | .7243 |
| point | 90 | is | .7248 |
| point | 91 | is | .7260 |
| point | 92 | is | .7273 |
| point | 93 | is | .7375 |
| point | 94 | is | .7416 |
| point | 95 | is | .7440 |
| point | 96 | is | .7502 |
| point | 97 | is | .7523 |
| point | 98 | is | .7541 |
| point | 99 | is | .7579 |
| point | 100 | is | .7732 |

SIMULATION OF GO-FLOW LIGHT BULB PROBLEM

| | | | | |
|---------|---------|---------|---|---|
| 20.00 | | | | |
| 0 | | | | |
| 1000 | | | | |
| 7 | | | | |
| 0 | | | | |
| 0. | | | | |
| 1.00 | | | | |
| 9.99 | | | | |
| 10.00 | | | | |
| 11.00 | | | | |
| 15.00 | | | | |
| 20.00 | | | | |
| 5 | | | | |
| BATTERY | passive | standby | 1 | 2 |
| .10000 | 0. | | | |
| 1.00000 | 1.00000 | 0. | | |
| | system | process | 0 | |
| | SWITCH1 | process | 0 | |
| | SWITCH2 | process | 0 | |
| SWITCH1 | switch | open | 3 | 1 |
| .30000 | 0. | | | |
| 1.00000 | 1.00000 | 0. | | |
| | system | command | 0 | |
| | system | power | 1 | |
| | BATTERY | process | 0 | |
| | LIGHT1 | process | 0 | |
| SWITCH2 | switch | open | 3 | 1 |
| .30000 | 0. | | | |
| 1.00000 | 1.00000 | 0. | | |
| | system | command | 0 | |
| | system | power | 1 | |
| | BATTERY | process | 0 | |
| | LIGHT2 | process | 0 | |
| LIGHT1 | passive | standby | 1 | 1 |
| .20000 | .00100 | | | |
| 1.00000 | 1.00000 | 0. | | |
| | SWITCH1 | process | 0 | |
| | system | process | 0 | |
| LIGHT2 | passive | standby | 1 | 1 |
| .20000 | .00100 | | | |
| 1.00000 | 1.00000 | 0. | | |
| | SWITCH2 | process | 0 | |
| | system | process | 0 | |
| standby | | | | |
| | 1 | | | |
| | 3 | | | |
| | 0. | 0 | | |
| | 1 | | | |
| system | BATTERY | process | | |
| | 1 | | | |
| | 0. | 0 | | |
| | 1 | | | |

```
system      SWITCH1    command
           -1
  10.00        0
           1
system      SWITCH2    command
           -1
```

AFTER 1000 TRIALS

THE TIME DEPENDENT UNAVAILABILITY IS AS FOLLOWS

| TIME | UNAVAILABILITY |
|-------|----------------|
| 0. | .5090 |
| 1.00 | .5090 |
| 9.99 | .5120 |
| 10.00 | .2940 |
| 11.00 | .2940 |
| 15.00 | .2990 |
| 20.00 | .3020 |

AFTER 1000 TRIALS
AND
OVER A TIME PERIOD OF 20 HOURS
THE AVERAGE SYSTEM UNAVAILABILITY IS AS FOLLOWS

The minimum is: .0000
The 1st percentile is: .0000
The 5th percentile is: .0000
The 25th percentile is: .0000
The 40th percentile is: .0000
The 50th percentile is: .0044
The 60th percentile is: .0492
The 75th percentile is: 1.0000
The 95th percentile is: 1.0000
The 99th percentile is: 1.0000
The maximum is: 1.0000
The mean is: .3416
The variance is: .2024

SIMULATION OF GO-FLOW LIGHT BULB PROBLEM

| | | | | |
|---------|---------|---------|---|---|
| 20.00 | | | | |
| 0 | | | | |
| 10000 | | | | |
| 7 | | | | |
| 0 | | | | |
| 0. | | | | |
| 1.00 | | | | |
| 9.99 | | | | |
| 10.00 | | | | |
| 11.00 | | | | |
| 15.00 | | | | |
| 20.00 | | | | |
| 5 | | | | |
| BATTERY | passive | standby | 1 | 2 |
| .10000 | 0. | | | |
| 1.00000 | 1.00000 | 0. | | |
| | system | process | 0 | |
| | SWITCH1 | process | 0 | |
| | SWITCH2 | process | 0 | |
| SWITCH1 | switch | open | 3 | 1 |
| .30000 | 0. | | | |
| 1.00000 | 1.00000 | 0. | | |
| | system | command | 0 | |
| | system | power | 1 | |
| | BATTERY | process | 0 | |
| | LIGHT1 | process | 0 | |
| SWITCH2 | switch | open | 3 | 1 |
| .30000 | 0. | | | |
| 1.00000 | 1.00000 | 0. | | |
| | system | command | 0 | |
| | system | power | 1 | |
| | BATTERY | process | 0 | |
| | LIGHT2 | process | 0 | |
| LIGHT1 | passive | standby | 1 | 1 |
| .20000 | .00100 | | | |
| 1.00000 | 1.00000 | 0. | | |
| | SWITCH1 | process | 0 | |
| | system | process | 0 | |
| LIGHT2 | passive | standby | 1 | 1 |
| .20000 | .00100 | | | |
| 1.00000 | 1.00000 | 0. | | |
| | SWITCH2 | process | 0 | |
| | system | process | 0 | |
| standby | | | | |
| | 1 | | | |
| | 3 | | | |
| 0. | | 0 | | |
| | 1 | | | |
| system | BATTERY | process | | |
| | 1 | | | |
| 0. | | 0 | | |
| | 1 | | | |

| | | |
|--------|---------|---------|
| system | SWITCH1 | command |
| | -1 | |
| | 10.00 | 0 |
| | 1 | |
| system | SWITCH2 | command |
| | -1 | |

AFTER 10000 TRIALS

THE TIME DEPENDENT UNAVAILABILITY IS AS FOLLOWS

| TIME | UNAVAILABILITY |
|-------|----------------|
| 0. | .4993 |
| 1.00 | .4999 |
| 9.99 | .5052 |
| 10.00 | .2757 |
| 11.00 | .2763 |
| 15.00 | .2787 |
| 20.00 | .2814 |

AFTER 10000 TRIALS
AND
OVER A TIME PERIOD OF 20 HOURS
THE AVERAGE SYSTEM UNAVAILABILITY IS AS FOLLOWS

The minimum is: .0000
The 1st percentile is: .0000
The 5th percentile is: .0000
The 25th percentile is: .0000
The 40th percentile is: .0000
The 50th percentile is: .0033
The 60th percentile is: .0289
The 75th percentile is: 1.0000
The 95th percentile is: 1.0000
The 99th percentile is: 1.0000
The maximum is: 1.0000
The mean is: .3225
The variance is: .1944

TEST OF THE TANK PORTION OF THE PROGRAM

| | | | | |
|---------|---------|---------|-----|---|
| | 1000.00 | | | |
| | 0 | | | |
| | 1000 | | | |
| | 201 | | | |
| | 1 | | | |
| | 3 | | | |
| unit1 | valve | open | 3 | 1 |
| | 0.0 | 0.00312 | | |
| | 1.0 | 1.0 | 0.0 | |
| | system | power | 1 | |
| | tank | process | 1 | |
| | tank | command | 1 | |
| | nowhere | process | 1 | |
| unit2 | valve | open | 3 | 1 |
| | 0.0 | 0.00456 | | |
| | 1.0 | 1.0 | 0.0 | |
| | system | power | 1 | |
| | system | process | 1 | |
| | tank | command | 1 | |
| | tank | process | 1 | |
| unit3 | valve | closed | 3 | 1 |
| | 0.0 | 0.0057 | | |
| | 1.0 | 1.0 | 0.0 | |
| | system | power | 1 | |
| | system | process | 1 | |
| | tank | command | -1 | |
| | tank | process | 0 | |
| standby | | | | |
| | 1 | | | |
| | 0 | | | |

AFTER 1000 TRIALS

THE TIME DEPENDENT UNAVAILABILITY ANALYSIS IS AS FOLLOWS

| TIME | UNAVAILABILITY |
|--------|----------------|
| 0. | 0. |
| 5.00 | 0. |
| 10.00 | 0. |
| 15.00 | 0. |
| 20.00 | 0. |
| 25.00 | .0010 |
| 30.00 | .0010 |
| 35.00 | .0040 |
| 40.00 | .0060 |
| 45.00 | .0100 |
| 50.00 | .0120 |
| 55.00 | .0130 |
| 60.00 | .0140 |
| 65.00 | .0160 |
| 70.00 | .0170 |
| 75.00 | .0200 |
| 80.00 | .0230 |
| 85.00 | .0240 |
| 90.00 | .0270 |
| 95.00 | .0300 |
| 100.00 | .0320 |
| 105.00 | .0330 |
| 110.00 | .0370 |
| 115.00 | .0400 |
| 120.00 | .0430 |
| 125.00 | .0460 |
| 130.00 | .0510 |
| 135.00 | .0580 |
| 140.00 | .0640 |
| 145.00 | .0690 |
| 150.00 | .0720 |
| 155.00 | .0740 |
| 160.00 | .0740 |
| 165.00 | .0760 |
| 170.00 | .0780 |
| 175.00 | .0830 |
| 180.00 | .0870 |
| 185.00 | .0870 |
| 190.00 | .0880 |
| 195.00 | .0920 |
| 200.00 | .0940 |
| 205.00 | .0950 |
| 210.00 | .1010 |
| 215.00 | .1020 |
| 220.00 | .1120 |
| 225.00 | .1160 |
| 230.00 | .1180 |
| 235.00 | .1210 |
| 240.00 | .1230 |
| 245.00 | .1260 |
| 250.00 | .1300 |

| | |
|--------|-------|
| 255.00 | .1350 |
| 260.00 | .1390 |
| 265.00 | .1430 |
| 270.00 | .1440 |
| 275.00 | .1500 |
| 280.00 | .1560 |
| 285.00 | .1570 |
| 290.00 | .1590 |
| 295.00 | .1630 |
| 300.00 | .1650 |
| 305.00 | .1670 |
| 310.00 | .1730 |
| 315.00 | .1770 |
| 320.00 | .1790 |
| 325.00 | .1800 |
| 330.00 | .1810 |
| 335.00 | .1830 |
| 340.00 | .1850 |
| 345.00 | .1850 |
| 350.00 | .1860 |
| 355.00 | .1890 |
| 360.00 | .1900 |
| 365.00 | .1920 |
| 370.00 | .1940 |
| 375.00 | .1970 |
| 380.00 | .2010 |
| 385.00 | .2020 |
| 390.00 | .2070 |
| 395.00 | .2100 |
| 400.00 | .2100 |
| 405.00 | .2100 |
| 410.00 | .2120 |
| 415.00 | .2140 |
| 420.00 | .2150 |
| 425.00 | .2170 |
| 430.00 | .2190 |
| 435.00 | .2210 |
| 440.00 | .2220 |
| 445.00 | .2240 |
| 450.00 | .2280 |
| 455.00 | .2290 |
| 460.00 | .2300 |
| 465.00 | .2340 |
| 470.00 | .2370 |
| 475.00 | .2370 |
| 480.00 | .2370 |
| 485.00 | .2400 |
| 490.00 | .2420 |
| 495.00 | .2420 |
| 500.00 | .2430 |
| 505.00 | .2470 |
| 510.00 | .2480 |
| 515.00 | .2500 |
| 520.00 | .2560 |
| 525.00 | .2570 |

| | |
|--------|-------|
| 530.00 | .2580 |
| 535.00 | .2600 |
| 540.00 | .2630 |
| 545.00 | .2630 |
| 550.00 | .2640 |
| 555.00 | .2640 |
| 560.00 | .2660 |
| 565.00 | .2670 |
| 570.00 | .2700 |
| 575.00 | .2720 |
| 580.00 | .2740 |
| 585.00 | .2750 |
| 590.00 | .2780 |
| 595.00 | .2790 |
| 600.00 | .2800 |
| 605.00 | .2800 |
| 610.00 | .2810 |
| 615.00 | .2820 |
| 620.00 | .2840 |
| 625.00 | .2850 |
| 630.00 | .2860 |
| 635.00 | .2880 |
| 640.00 | .2890 |
| 645.00 | .2900 |
| 650.00 | .2920 |
| 655.00 | .2930 |
| 660.00 | .2940 |
| 665.00 | .2940 |
| 670.00 | .2950 |
| 675.00 | .2970 |
| 680.00 | .2990 |
| 685.00 | .3010 |
| 690.00 | .3020 |
| 695.00 | .3060 |
| 700.00 | .3070 |
| 705.00 | .3090 |
| 710.00 | .3090 |
| 715.00 | .3090 |
| 720.00 | .3090 |
| 725.00 | .3100 |
| 730.00 | .3100 |
| 735.00 | .3110 |
| 740.00 | .3130 |
| 745.00 | .3160 |
| 750.00 | .3170 |
| 755.00 | .3180 |
| 760.00 | .3180 |
| 765.00 | .3200 |
| 770.00 | .3210 |
| 775.00 | .3210 |
| 780.00 | .3210 |
| 785.00 | .3210 |
| 790.00 | .3220 |
| 795.00 | .3220 |
| 800.00 | .3220 |

| | |
|---------|-------|
| 805.00 | .3230 |
| 810.00 | .3240 |
| 815.00 | .3240 |
| 820.00 | .3250 |
| 825.00 | .3250 |
| 830.00 | .3250 |
| 835.00 | .3250 |
| 840.00 | .3260 |
| 845.00 | .3260 |
| 850.00 | .3260 |
| 855.00 | .3260 |
| 860.00 | .3260 |
| 865.00 | .3260 |
| 870.00 | .3270 |
| 875.00 | .3270 |
| 880.00 | .3270 |
| 885.00 | .3270 |
| 890.00 | .3270 |
| 895.00 | .3290 |
| 900.00 | .3300 |
| 905.00 | .3310 |
| 910.00 | .3320 |
| 915.00 | .3330 |
| 920.00 | .3330 |
| 925.00 | .3340 |
| 930.00 | .3340 |
| 935.00 | .3350 |
| 940.00 | .3350 |
| 945.00 | .3350 |
| 950.00 | .3350 |
| 955.00 | .3360 |
| 960.00 | .3360 |
| 965.00 | .3360 |
| 970.00 | .3360 |
| 975.00 | .3360 |
| 980.00 | .3360 |
| 985.00 | .3370 |
| 990.00 | .3370 |
| 995.00 | .3380 |
| 1000.00 | .3380 |

AFTER 1000 TRIALS

THE UNAVAILABILITY DISTRIBUTION DATA IS AS FOLLOWS

| | |
|-------------------------|-------|
| The minimum is: | .0000 |
| The 1st percentile is: | .0000 |
| The 5th percentile is: | .0000 |
| The 25th percentile is: | .0000 |
| The 40th percentile is: | .0000 |
| The median is: | .0000 |
| The mean is: | .2155 |
| The 60th percentile is: | .0000 |
| The 75th percentile is: | .4840 |
| The 95th percentile is: | .8701 |
| The 99th percentile is: | .9540 |
| The maximum is: | .9790 |
| The variance is: | .1085 |